View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Digital library of Brno University of Technology

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ANALÝZA POKRYTÍ TESTY JBOSS APLIKAČNÍHO SERVERU

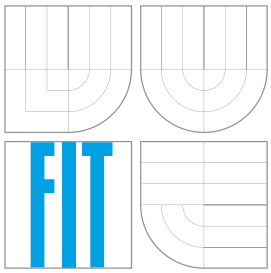
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

NIKOLETA ŽIAKOVÁ

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ANALÝZA POKRYTÍ TESTY JBOSS APLIKAČNÍHO SERVERU

TESTS CODE COVERAGE OF JBOSS APPLICATION SERVER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — NIKOLETA ŽIAKOVÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZDENĚK LETKO
SUPERVISOR

BRNO 2011

Abstrakt

Tato práce se zabývá analýzou pokrytí kódu testovací sadou aplikačního serveru JBoss. Pro sběr informací o pokrytí kódu byly použity nástroje Emma a Cobertura. Bakalářská práce je rozdělená na dvě hlavní části. První z nich seznámí čtenáře s použitými technologiemi a připraví teoretický základ pro druhou část. Ta má praktický charakter, obsahuje postup pro získání informací o pokrytí kódu vybranými nástroji a jejich následnou analýzu. Výsledky poskytují pohled na vývoj pokrytí kódu v průběhu vývoje aplikačního serveru a také porovnání použitých nástrojů.

Abstract

This thesis deals with tests code coverage of JBoss application server. To gather code coverage information, the tools Emma and Cobertura were used. The bachelor's thesis is divided into two main parts. The first of them acquaints reader with technologies used and prepares theoretical basis for the second part. The second part is practical and describes procedure for obtaining coverage information by chosen tools and its subsequent analysis. Results provide an overview of code coverage development during development of the applicaion server as well as a comparison of the used tools.

Klíčová slova

JBoss AS, Emma, Cobertura, pokrytí kódu, metriky, testovací sada, Java EE

Keywords

JBoss AS, Emma, Cobertura, code coverage, metrics, testsuite, Java EE

Citace

Nikoleta Žiaková: Analýza pokrytí testy JBoss aplikačního serveru, bakalářská práce, Brno, FIT VUT v Brně, 2011

Analýza pokrytí testy JBoss aplikačního serveru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Zdeňka Letka

.....
Nikoleta Žiaková
12. mája 2011

Poděkování

Ráda bych poděkovala vedoucímu této práce Zdeňkovi Letkovi a technickému vedoucímu Martinovi Večeřovi za jejich čas, trpělivost a cenné rady, které mi při řešení bakalářské práce poskytli.

© Nikoleta Žiaková, 2011.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Testovanie je neodmysliteľnou súčasťou vývoja softwaru. Vykonáva sa, pretože programy sú rovnako nedokonalé ako ľudia, ktorí ich píšu. Samotný proces testovania síce môže chyby odhaliť, ale nemôže nám povedať, že v programe sa už ďalšie nevyskytujú. Aby testovanie splnilo svoj účel a zároveň bolo efektívne, je potrebné sa venovať aj kvalite samotných testov.

Kód, ktorý sa počas testov vôbec nevykonáva, nemôže byť ani testovaný. Nedostatok testovania niektorej časti kódu automaticky vedie k nedokonalému testovaniu. Na odhalenie takýchto častí kódu slúži analýza pokrytia kódu. Pokrytie kódu je metrika, ktorá vyjadruje, aké množstvo zdrojového kódu aplikácie je vykonané počas testov, inými slovami – ako dobre je daná aplikácia testovaná. [17]

Témou tejto bakalárskej práce je analýza pokrytia kódu aplikačného servera JBoss. JBoss AS má vlastnú testovaciu sadu, ktorá obsahuje veľké možstvo testov. Otázkou však je, či sú tieto testy napriek svojmu rozsahu dostatočné. Analýze predchádza proces zberu informácií o pokrytí kódu, ktorý zahŕňa výber vyhovujúceho nástroja, začlenenie zberu dát do testovacieho procesu JBoss AS a následnú vhodnú reprezentáciu nazbieraných dát. Cieľom práce je tento proces navrhnúť a realizovať.

Jadro práce je rozdelené na dve hlavné časti, z ktorých každá predstavuje jednu kapitolu. V prvej z nich predstavím technológie, ktoré som pri práci použila, a teda pripravuje teoretický základ pre zvyšok práce. Vysvetlím tu pojmy ako Java Enterprise Edition, testovanie softwaru či analýza pokrytia kódu. Stručne predstavím aj aplikačný server JBoss a aplikáciu Ant, s ktorými budem ďalej pracovať.

Druhá časť je zameraná prakticky. Krok po kroku v nej uvediem postup, ktorým som získala výsledky pokrytia kódu dvomi nástrojmi. Vybranými nástrojmi sú Emma a Cobertura. Tieto výsledky potom budem analyzovať z rôznych pohľadov. Analýza bude zameraná na vývoj pokrytia kódu v priebehu vývoja aplikačného servera JBoss, použité nástroje a metriky pokrytia kódu. Použitie dvoch nástrojov mi umožní porovnať výsledky, ktoré poskytujú. Pozriem sa aj na testovaciu sadu aplikačného servera JBoss a na možnosť jej optimalizácie s ohľadom na hodnoty pokrytia kódu.

Kapitola 2

Použité technológie

Táto kapitola predstavuje použité technológie, ktoré sú potrebné pre pochopenie celej bakalárskej práce. Najprv sa budem venovať platforme Java Enterprise Edition, pretože tvorí základ aplikačného servera JBoss. Popis programovacieho jazyka Java, z ktorého táto platforma vychádza, tu neuvádzam, je možné ho nájsť napr. v knihe [4]. V ďalšej podkapitole predstavím aplikačný server JBoss a potom venujem krátku podkapitolu aplikácii Ant. Nasledujúca časť sa zaoberá testovaním softwaru, jeho metódami a najčastejšími prístupmi k testovaniu v jazyku Java. Posledná podkapitola je venovaná analýze pokrytia kódu testovacími prípadmi, ktorá je témou tejto bakalárskej práce.

2.1 Platforma Java Enterprise Edition

Platforma Java Enterprise Edition (Java EE alebo J2EE) [14] je jednou z edícií platformy Java. Ďalšími edíciami sú Java Platform, Standard Edition (Java SE)[1] používaná prevažne v klasických desktopových aplikáciách a Java Platform, Micro Edition (Java ME)[2] pre mobilné (angl. embedded) zariadenia.

Enterprise, teda podnikové aplikácie, majú väčší rozsah ako bežné desktopové aplikácie, môžu bežať na viacerých počítačoch, sú schopné obslúžiť veľké množstvo užívateľov naraz, komunikujú s viacerými systémami a môžu zahŕňať niekoľko aplikácií. Takéto aplikácie môžu mať niekoľko užívateľských rozhraní, napr. webové rozhranie pre zákazníkov a grafické užívateľské rozhranie aplikácie bežiacej na kancelárskom počítači.

Java EE aplikácie sú dodávané v Java Archive (JAR), Web Archive (WAR) alebo Enterprise Archive (EAR) [6] súboroch. Takéto balenie aplikácií umožňuje zostaviť množstvo Java EE aplikácií s využitím rovnakých komponentov bez nutnosti prepisovať ich zdrojový kód.

2.1.1 Viacvrstvová architektúra

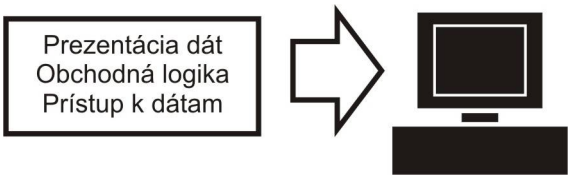
Podnikové aplikácie sa často skladajú z troch logických častí, z ktorých každá má inú úlohu.

- Prezentačná – časť poskytuje prezentáciu dát užívateľovi a spôsoby prezentácie dát od užívateľa systému.
- Riadiaca časť – prijíma a spracováva požiadavky od prezentačnej vrstvy a posiela naspäť výsledky.

[1] http://download.oracle.com/javase/
[2] http://download.oracle.com/javame/

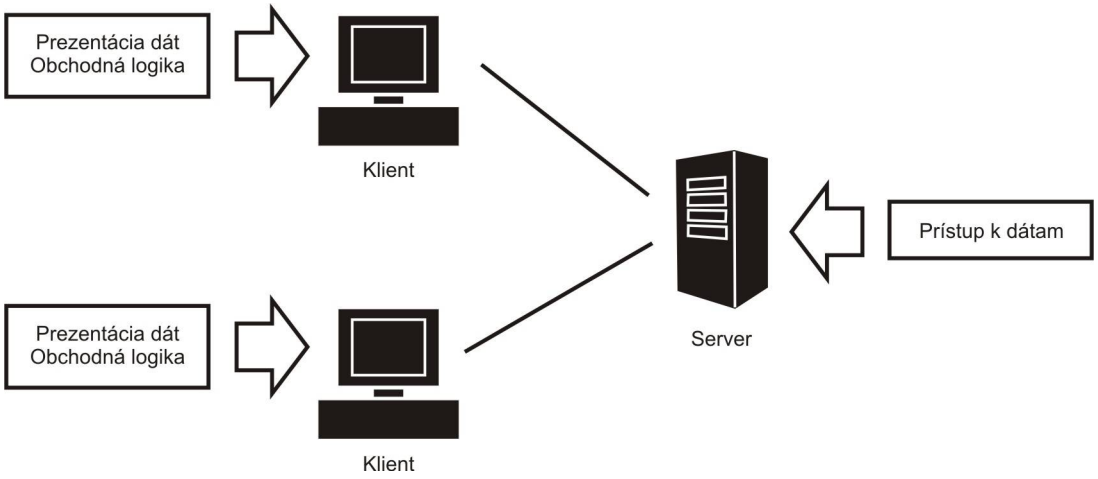
- Dátová časť – je zodpovedná za ukladanie a načítavanie dát, pričom môže používať ľubovoľný zdroj (napríklad databázu).

Jednoduché aplikácie, ktoré sú určené pre beh na jednom počítači a poskytujú služby všetkých spomenutých vrstiev (užívateľské rozhranie, spracovanie, uloženie dát), sa nazývajú *jednovrstvové systémy* (Obrázok 2.1). Tieto systémy sú ľahko spravovateľné a keďže sú všetky dáta uložené na jednom mieste, nevzniká problém nekonzistencie dát. Má to však aj svoje nevýhody, a to nemožnosť súčasného prístupu viacerých užívateľov k aplikácii.



Obrázok 2.1: Jednovrstvová architektúra.

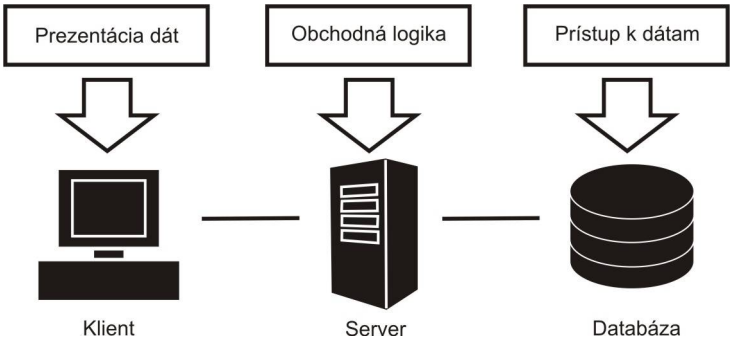
Tento problém rieši dvojvrstvový systém, tzv. *klient-server architektúra*, ktorá je založená na komunikácii medzi klientom a serverom. Databázový server poskytuje ukladanie perzistentných dát. V tomto prípade je časť aplikácie pre prístup k dátam oddelená od zvyšku aplikačnej logiky. Databáza môže bežať v oddelenom procese alebo dokonca na inom fyzickom počítači. K serveru môžu naraz pristupovať viacerí klienti, ako je naznačené na Obrázku 2.2.



Obrázok 2.2: Architektúra klient-server.

V *trojvrstvovej architektúre* (Obrázok 2.3) je navyše oddelená riadiaca vrstva, ktorá tvorí prepojenie pracovnej stanice s databázou a implementuje aplikačnú logiku. Keďže aj táto časť beží na serveri, je prístupná ľubovoľnému počtu užívateľov. Pri zvyšovaní počtu užívateľov požadujúcich služby riadiacej vrstvy a rozširovaní aplikačnej logiky je možné počet serverov zvýšiť. Klientska aplikácia prestáva byť zodpovedná za prístup k dátam a presadzovanie obchodných pravidiel. Jej úlohou ostáva prezentácia užívateľského rozhrania a komunikácia s riadiacou vrstvou.

Proces delenia aplikačnej logiky môže pokračovať na ľubovoľnú úroveň n-vrstvovej architektúry. Viacvrstvové aplikácie sú ľahšie udržovateľné, flexibilné, poskytujú konzistenciu



Obrázok 2.3: Trojvrstvová architektúra.

dát, spoluprácu medzi viacerými aplikáciami a poskytujú väčšiu bezpečnosť. N-vrstvová architektúra nevyžaduje prísne beh každej aplikačnej vrstvy na samostatnom stroji, ale kladom je, že to dovoľuje.

Java EE definuje štandardy pre implementáciu aplikácií pozostávajúcich z dvoch, troch alebo viacerých vrstiev a poskytuje prostriedky na ich prepojenie. Distribuovaná architektúra Java EE je znázornená na Obrázku 2.4. Klientska vrstva beží na klientskom počítači, webová a business vrstva bežia na Java EE serveri a dátová vrstva beží na databázovom serveri. Napriek tomu, že sa táto architektúra skladá zo štyroch vrstiev, nazýva sa trojvrstvová, pretože je zvyčajne umiestnená (distribuovaná) na troch oddelených miestach.

2.1.2 Java EE komponenty

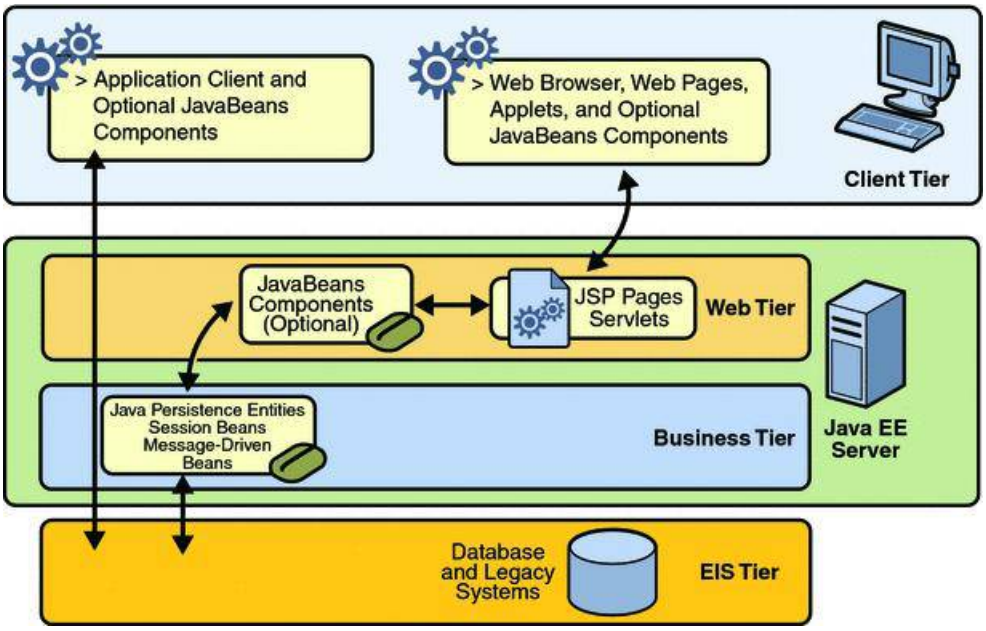
Logika Java EE aplikácií je rozdelená podľa funkcie na samostatné softwarové celky, ktorým sa hovorí komponenty [14]. Java EE komponenty sú napísané v programovacom jazyku Java podľa Java EE štandardu. Od bežných Java tried sa odlišujú tým, že je možné ich nasadiť do aplikačného servera. Java EE špecifikácia určuje tieto Java EE komponenty:

- klientske komponenty,
- webové komponenty,
- enterprise komponenty.

Konkrétne komponenty zaradené do jednotlivých vrstiev Java EE architektúry sú znázornené na Obrázku 2.4.

Do klientskej vrstvy radíme typicky aplikačných klientov ako internetový prehliadač, applety, ale môže to byť v podstate ľubovoľná Java aplikácia. Aplikačný klient beží na klientskom počítači a môže poskytovať bohatšie užívateľské rozhranie, ako môže byť vytvorené značkovacími jazykmi. Priamo pristupuje k enterprise komponentom business vrstvy a cez hypertext transfer protokol (HTTP) [7] môže komunikovať so servletmi webovej vrstvy. Je to typicky tenký klient, teda nevykonáva nijakú aplikačnú logiku. Applet je malá klientska aplikácia, ktorá beží v JVM [13] vo webovom prehliadači.

Medzi webové komponenty patria Java Servlet, JavaServer Faces a JavaServer Pages [14]. Java Servlet je technológia na rozšírenie webového servera, ktorá slúži na generovanie stránok s dynamickým obsahom. Servlet je vyvolaný ako odpoveď na požiadavku od klienta. JavaServer Faces poskytuje aplikačné programové rozhranie (API) pre tvorbu



Obrázok 2.4: Vrstvy Java EE architektúry. Prevzaté z [5].

užívateľských rozhraní na strane servera. JavaServer Pages (JSPs) slúžia podobne ako servlety na tvorbu dynamických stránok. JSP stránka obsahuje statické dáta zapísané pomocou značkovacieho jazyka a JSP prvky, ktoré určujú dynamický obsah.

Enterprise komponentmi sú Enterprise Java Beans (EJBs) [14] – vlajková loď Java EE platformy. EJB sú komponenty, ktoré implementujú business logiku, teda hlavnú funkciu Java EE aplikácie. EJB prijíma dáta od klienta, spracováva ich a posiela na uloženie integračnej vrstve a naopak. Existujú rôzne typy EJBs: session beans, message-driven beans a entity beans.

2.1.3 Java EE kontajnery a aplikačný server

Java EE poskytuje ku každému Java EE komponentu základnú službu vo forme kontajnera [14]. Kontajnery sa starajú o bežné detaily ako spustenie služieb na strane servera, aktivovanie aplikačnej logiky, odstránenie komponentu. Kontajnery riadia konfigurovateľné (napr. bezpečnostné nastavenia) aj nekonfigurovateľné služby (napr. perzistencia dát, životný cyklus komponentov), ktoré Java EE poskytuje. Konfigurovateľné služby môžu spôsobiť rozdielne správanie Java EE aplikácie v závislosti od toho, v akom prostredí je nasadená.

Rozlišujeme kontajnery na klientskom počítači a kontajnery aplikačného servera. Aplikačný server poskytuje nasledujúce kontajnery pre webové a business komponenty:

- *Enterprise JavaBeans kontajner* – riadi vykonávanie a životný cyklus EJB komponentov.
- *Webový kontajner* – spravuje vykonávanie a životný cyklus webových komponentov.

Na klientskom počítači sú tieto kontajnery:

- *Kontajner aplikačného klienta* – zabezpečuje beh aplikačného klienta.

- *Kontajner appletov* – stará sa o beh appletov na klientovi, skladá sa z webového prehliadača a Java zásuvného modulu.

Aplikačný server (AS) [14, 10] tvorí prostredie pre beh Java EE produktu. Je to vlastne rozhranie medzi operačným systémom a podnikovou aplikáciou. Existujú komerčné aj open source aplikačné servery. Aplikačný server JBoss bude bližšie popísaný v nasledujúcej podkapitole 2.2. Medzi ostatné Java EE aplikačné servery patria:

- *Open source* – Apache Geronimo, Glassfish AS, Apache Tomcat a iné.
- *Komerčné* – WebLogic (Oracle), WebSpehere AS (IBM) a iné.

2.2 Aplikačný server JBoss

JBoss AS [10] je open source Java EE aplikačný server. To znamená, že je to implementácia enterprise časti platformy Java, ktorá pokrýva vyššie spomenuté služby ako JSP, EJB, servlety a ďalšie. Keďže je napísaný v programovacom jazyku Java, je prenositeľný na všetky operačné systémy, na ktorých funguje Java.

Podľa knihy *JBoss in action* [10] vznikol JBoss ako malý projekt v roku 1999, ktorého autorom bol Marc Fleury. Pokrýval implementáciu EJB časti Java EE špecifikácie. O dva roky neskôr, vo verzii JBoss AS 3, sa už stal plnohodnotným Java EE aplikačným serverom. JBoss AS 4 implementuje Java EE 1.4 a JBoss AS 5 vystupuje ako Java EE 5 aplikačný server. Aktuálna stabilná verzia JBoss AS 6 sa ďalej nevyvíja, namiesto nej v súčasnosti prichádza AS 7 s úplne novou architektúrou.

2.2.1 Konfigurácie JBoss AS a nasadzovanie aplikácií

Konfigurácia [10] aplikačného servera je množina služieb a aplikácií, ktoré sú pri spustení servera spustené. JBoss AS poskytuje tri základné konfigurácie:

- *default* – je implicitná konfigurácia, zahŕňa všetky služby nevyhnutné pre spustenie servera bez služieb clustering services.
- *minimal* – spúšťa minimálnu množinu služieb vrátane mikrokontajnera a služby JNDI.
- *all* – spúšťa všetky služby, ktoré JBoss AS poskytuje.

Tieto konfigurácie si môžeme prispôsobovať alebo pridávať vlastné konfigurácie. Často sa využíva zmenšovanie servera odstránením nadbytočných služieb, tzv. *slimming* [18]. Takto je možné spustiť len nevyhnutné služby, čím sa znižujú pamäťové nároky a zvyšuje rýchlosť. Spustenie menšieho počtu služieb tiež prináša výhodu menšej náchylnosti k poruchám, najmä z hľadiska bezpečnosti.

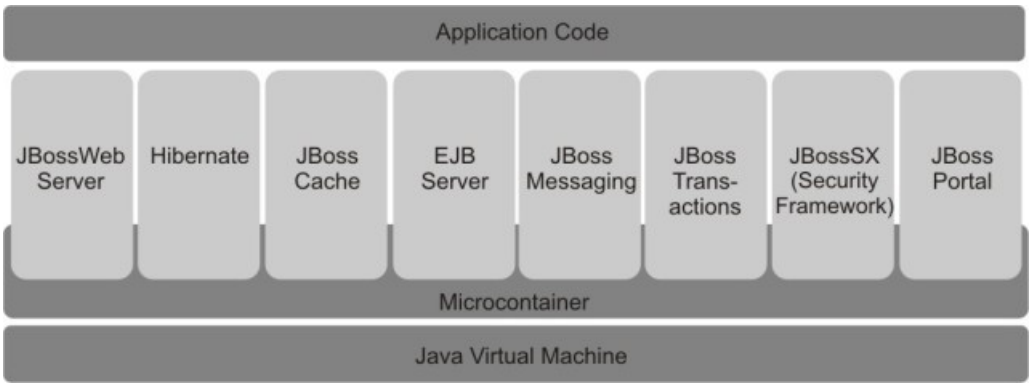
Všetky súbory potrebné pre spustenie JBoss AS sú umiestnené v jedinom adresári. Jedným z jeho podadresárov je adresár *server/*, ktorý obsahuje dostupné konfigurácie. Každá konfigurácia má svoj adresár *deploy/*, kde sú nasadzované aplikácie a služby. Nasadenie aplikácie spočíva v jej skopírovaní do tohto adresára. Aplikácie môžu byť nasadené pomocou tzv. *cold deployment* [18], kedy je potrebné JBoss AS vypnúť, aplikáciu nasadiť a server znova spustiť. Ďalším spôsobom je *hot deployment* [18], pri ktorom je možné aplikáciu nasadiť za behu aplikačného servera.

2.2.2 Architektúra JBoss AS

JBoss AS je kolekciou nezávislých komponentov, z ktorých každý sa zameriava na konkrétnu oblasť Java EE funkcionality. Prvé verzie JBoss AS boli postavené na jadre Java Management Extension (JMX) [18], ktoré poskytovalo základnú funkcionalitu. Všetky služby dodávané aplikačným serverom boli písané ako Managed Beans (MBeans) [18] a pripojovali sa k JMX jadru. Táto architektúra dávala možnosť jednoducho pridávať nové služby a naopak, odoberať nepotrebné.

Od verzie 4.0.3 začal JBoss AS prechádzať na architektúru mikrokontajnera [18] (Obrázok 2.5), ktorý umožňuje implementovať nové služby použitím Plain Old Java Objects (POJOs) namiesto MBeans. Mikrokontajnerová architektúra je oproti JMX jadru oveľa jednoduchšia, pretože nemusí podporovať JMX, čo znamená, že je možné zostaviť ešte menšiu minimálnu konfiguráciu. Ďalšou výhodou je, že služby postavené na báze mikrokontajnera môžu byť nasadené samostatne, dokonca aj v rámci iného aplikačného servera (ako WebLogic Server, Tomcat).

JMX jadro má stále významnú úlohu v architektúre JBoss AS, pretože nie všetky služby boli prenesené do mikrokontajnera. Časom by sa to však malo zmeniť a všetky služby by mali závisieť výlučne na mikrokontajneri. JMX jadro prestáva byť hlavnou architektúrou aplikačného servera a stáva sa jednou zo služieb nasadených do mikrokontajnera, ktorá spravuje a monitoruje komponenty.



Obrázok 2.5: Architektúra JBoss AS. Prevzaté z [18].

Podobne ako do JMX jadra, aj do mikrokontajnera sú pridávané komponenty podľa potreby aplikácie. Mikrokontajner zabezpečuje komunikáciu komponentov s JVM a jednotlivých komponentov navzájom. Kód aplikácie nasadený v aplikačnom serveri potom využíva rôzne služby pripojené do mikrokontajnera. Medzi tieto služby patria:

- *JBoss Web Server* – integrovaný webový server, ktorý dovoľuje použitie webových technológií ako servlet, JavaServer Pages, JavaServer Faces.
- *Hibernate* – je nástroj, ktorý umožňuje objektovo-relačné mapovanie, teda zabezpečuje perzistenciu objektov podľa EJB3 [18] špecifikácie.
- *JBoss Cache* – transakčná, distribuovaná rýchla vyrovnávacia pamäť používaná mnohými ďalšími službami JBoss AS.
- *EJB Server* – implementácia EJB3 špecifikácie.

- *JBoss Messaging* – server pre posielanie správ podľa JSR-168 špecifikácie [10]. Umožňuje synchrónnu aj asynchrónnu komunikáciu.
- *JBossSX* – deklaratívna bezpečnostná služba založená na úlohách.
- *JBoss Portal* – portálový server implementujúci JSR-168 špecifikáciu. Portál slúži na vytvorenie webových stránok zložením rôznorodých kúskov zdrojového kódu – portletov [18], ktoré navonok pracujú ako jedna aplikácia.

Detaily o týchto a ďalších službách JBoss AS je možné nájsť napr. v knihe [18].

2.3 Ant

Aplikácia Ant [8] slúži na automatický preklad programov zo zdrojového do cieľového kódu. Jeho výhodou je, že je nezávislý na operačnom systéme, pretože je napísaný v programovacom jazyku Java. Na popis zostavovacieho procesu používa jazyk XML [19].

Implicitným názvom zostavovacieho súbora je build.xml, ktorý sa dá zmeniť spustením aplikácie s parametrom `-buildfile`, `-file` alebo `-f`. Jeho koreňový element je element *project*, a má tri atribúty:

- *name* – nepovinný parameter, ktorý definuje meno zostavovaného projektu,
- *basedir* – bázový adresár pre celý zostavovací súbor,
- *default* – meno implicitného cieľa, ktorý sa prevedie pri spustení aplikácie bez parametra.

Na riadenie prekladu sa používajú *ciele* definované elementom *target*. Závislosti medzi jednotlivými cieľmi umožňuje definovať atribút *depends*, od ktorého potom závisí poradie, v akom sa budú vykonávať. Telo elementu *target* sa skladá z *úloh* (task), ktoré majú byť v rámci daného cieľa prevedené. Úlohy sú dodávané s aplikáciou Ant alebo v špeciálnych balíčkoch, ale je možné implementovať aj vlastné úlohy.

Ďalším dôležitým elementom je *property*, ktorý nastavuje vlastnosti v zostavovacom súbore. Vlastnostiam, ktoré majú úlohu akýchsi konštánt, môžeme nastaviť meno (atribútom *name*) a hodnotu (atribútom *value*). Definícia celej množiny vlastností sa môže nachádzať aj v externom súbore, odkiaľ sa načítajú pomocou atribútu *resource* alebo *file*.

Na definovanie zoznamu ciest slúžia elementy *path* a *classpath*. Do zoznamu je možné pridať prvok prostredníctvom atribútu *location* alebo sa atribútom *refid* odkázať na cestu definovanú inde. Podobnú funkciu majú aj elementy *dirset* a *fileset*, ktoré naviac ponúkajú možnosť zahrnúť alebo nezahrnúť do zoznamu jednotlivé adresáre a súbory podľa toho, či vyhovujú danému vzoru. Vzory sa nachádzajú v atribútoch *includes/excludes* alebo *includesfile/excludesfile*.

2.4 Testovanie softwaru

Testovanie [9] je jednou z etáp životného cyklu softwaru. Je to proces, ktorý poskytuje na rozumnej úrovni istotu vo vytvorenom systéme. V najjednoduchšom zmysle je to overovanie, že všetky časti systému do seba zapadajú a výsledný produkt je to, čo sme chceli na začiatku. Testovanie je teda vyhodnotenie, že sme postavili systém správne a že

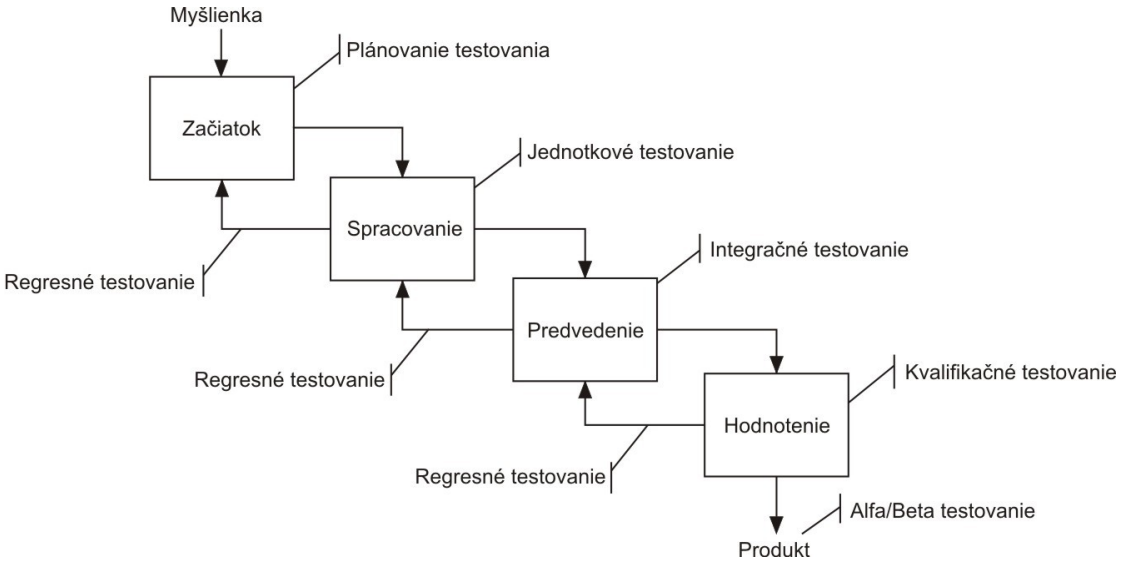
sme postavili správny systém. Tieto dve rozdielne činnosti sa nazývajú verifikácia (systém je postavený správne) a validácia (postavili sme to, čo sme chceli postaviť). Zlyhanie ktorejkoľvek časti je v rámci vývoja softwaru neprijateľné.

Pri testovaní sa zvyčajne hľadajú nasledujúce vlastnosti systému:

- *Správnosť* – systém vykonáva svoju funkciu správne. Toto je základom väčšiny testov.
- *Úplnosť* – systém robí všetko, čo je od neho očakávané.
- *Poruchy* – systém vie, keď zlyhá. Každá porucha musí byť rozpoznaná a systém musí zareagovať adekvátne. V prípade poruchy sa systém nemôže zrútiť. Aj napriek tomu, že táto vlastnosť nebýva špecifikovaná v požiadavkách, je považovaná za samozrejmosť.
- *Istota* – systém je hotový. Toto je kľúčovým prvkom testovania.

Existujú dve základné metódy testovania, a to *metóda bielej skrinky* (white box) a *metóda čiernej skrinky* (black box). Testovanie metódou bielej skrinky prebieha so znalosťou vnútornej štruktúry testovanej časti. Táto metóda často odhalí viac chýb v systéme vďaka schopnosti vidieť aktuálnu implementáciu. Na druhú stranu testy metódy čiernej skrinky sú navrhnuté tak, aby prebiehali bez ohľadu na vnútornú časť. Tieto testy dokážu identifikovať najkritickejšie chyby systému. Napriek protichodnosti týchto metód je nutné používať ich spolu pre dôkladné otestovanie systému a odhalenie jeho závad.

Jednou z dôležitých otázok pri tvorbe plánu vývoja softwaru je, kedy by testovanie malo prebiehať. V prípade, že začneme príliš skoro, riskujeme, že vytvoríme testy na podsystémy vo vývoji. Naopak, pri testovaní na konci vývojového cyklu budeme mať príliš veľký systém, ktorý treba pokryť. Preto je potrebné zaradiť testovanie v priebehu procesu vývoja softwaru na rozdielnych úrovniach. Takto prebieha testovanie zdola nahor. V skorých fázach projektu sa zameriavame na nízkoúrovňové časti systému a postupne prechádzame k rozhraniam a záležitostiam vyššej úrovne až po funkčnosť systému z hľadiska užívateľa. Proces testovania v závislosti od fázy vývoja softwaru je znázornený na Obrázku 2.6.



Obrázok 2.6: Testovanie v rôznych fázach projektu.

Projekt prejde všetkými fázami bez ohľadu na model životného cyklu softwaru, na základe ktorého projekt vyvíjame, preto aj testovanie by malo prejsť všetkými uvedenými fázami:

- *Jednotkové testovanie* [9] – je softwarové testovanie na najnižšej úrovni, kedy sú testované jednotlivé triedy a metódy, dôraz je kladený na verifikáciu.
- *Integračné testovanie* [9] – zahŕňa overovanie vzájomnej súčinnosti jednotlivých častí systému.
- *Kvalifikačné testovanie* [9] – je vrcholom jednotkového a integračného testovania, zameranie sa presúva k validácii systému. Testy sú koncipované tak, aby sa venovali požiadavkám na systém a jeho budúcemu použitiu.
- *Alfa/Beta testovanie* [9] – vykonáva zákazník na reálnych dátach. Alfa testovanie prebieha v sídle vývojového tímu, beta testovanie u užívateľa.
- *Regresné testovanie* [9] – zahŕňa opakovaný beh a prehodnotenie testov alebo ich podmnožiny pri každej zmene v systéme. Kým sa systém mení, vždy je tu možnosť zavedenia novej chyby. Často je potrebné začleniť nové testy.

V nasledujúcich podkapitolách budú bližšie popísané prístupy k testovaniu softwaru v jazyku Java.

2.4.1 Jednotkové testovanie

Jednotkový test (unit test) je test navrhnutý na vyhodnotenie jedného prvku kódu. Správanie kódu je vyšetrované nezávisle na ostatných prvkoch, aby sme vedeli rozhodnúť, či tento prvok spĺňa očakávania. Dôležité je, aby algoritmy pracovali správne a všetky triedy splnili požiadavky, ktoré na ne máme.

Tento typ testovania je často prevádzaný neformálnym spôsobom, napr. popretkávaním zdrojového kódu rôznymi výpismi, vytvorením metódy main(), skompilovaním a následným spustením v priebehu vývoja. Táto metóda je efektívna pri individuálnom programovaní, ale môže viesť k nesprávnemu testovaniu.

Vhodné jednotkové testy môžu výrazne zvýšiť kvalitu a spoľahlivosť kódu, obzvlášť v spojení s modernými agilnými metódami *extrémneho programovania* [1] (eXtreme Programming, XP). Jadrom XP je *programovanie riadené testami* [1] (test-driven development, TDD). Pri TDD programátor najprv napíše jednotkový test, ktorý otestuje určitú funkcionalitu, až potom implementuje samotnú funkcionalitu najjednoduchším možným spôsobom, aby test uspel. To znamená, že v systéme neexistuje kód, ktorý nie je otestovaný.

V objektovo-orientovanom programovaní existujú dva typy jednotiek, ktoré sa používajú pri jednotkovom testovaní, a to trieda a metóda. Trieda pozostáva z atribútov a metód, metóda predstavuje správanie triedy. Je preto logické predpokladať, že jednotkové testovanie na úrovni tried úspešne prevedie potreby jednotkového testovania. Avšak v niektorých prípadoch môže byť vhodné testovanie na úrovni metód.

Pri vytváraní testovacieho frameworku pre projekt je veľmi žiaduce vyhodnotiť tieto dve možnosti a rozhodnúť sa, ktorú budeme používať. Pri snahe zjednotiť testovanie na úrovni tried aj metód v jednom projekte sa spravovanie testovacej sady stáva čoraz viac zložitým. Je tiež veľmi pravdepodobné, že skončíme s nadbytočnými testami a/alebo budeme mať veľké časti kódu, ktoré nie sú testami vôbec pokryté.

Existuje mnoho frameworkov pre jednotkové testy v rôznych programovacích jazykoch, súhrnne sú nazývané xUnit. Pre programovací jazyk Java je to framework JUnit [8], ktorý uľahčuje písanie a spúšťanie automatizovaných jednotkových testov.

Testovanie na úrovni tried

Ako názov naznačuje, testovanie na úrovni tried [9] zahŕňa vytvorenie testovacieho prípadu pre každú triedu. Tento testovací prípad ohodnotí členov triedy, či ich správanie korešponduje s návrhom. Medzi dôvody použitia jednotkového testovania na úrovni tried patria:

- jednoduché začlenenie do procesu zostavenia a nasadenia,
- poskytuje dokumentáciu plánovaného správania triedy,
- môže byť použité pre regresné testy pri zmenách,
- oproti testovaniu na úrovni metód poskytuje lepší vysokoúrovňový prehľad systému a je jednoduchšie na udržovanie,
- poskytuje prehľad o pokroku pre zákazníkov a manažérov.

Rozoznávame dva druhy testovania na úrovni tried: testovanie implementácie a testovanie dedičnosti. Testovanie v zmysle plnej dedičnosti sa nazýva ploským testovaním. Testovanie triedy v zmysle implementácie znamená testovanie len metód implementovaných danou triedou alebo jedným z jej objektov, nie metód, ktorých implementácia je zdedená z nadradených tried.

Testovanie na úrovni metód

Testovanie na úrovni metód [9] pozostáva z návrhu testovacieho prípadu pre beh jedinej metódy a následného návrhu testovacej sady pre beh viacerých testovacích prípadov spolu. Testovanie na úrovni metód je vhodné pri existencii malého jadra tried, ktoré vykonávajú podstatnú časť práce. Môže byť veľmi užitočné pri hodnotení robustnosti, teda ako sú spracované neočakávané situácie. Tieto testy môžu byť použité napr. na overenie predaných argumentov alebo hodnôt.

2.4.2 Integračné testovanie

Ako vývoj postupuje, triedy a metódy sú spájané do logických a fyzických celkov. Zameranie sa posúva z práce jednotlivých metód a činnosti algoritmov na interakcie medzi triedami, robustnosť, ošetrovanie chýb a celkovú komunikáciu viacerých komponentov. V tejto fáze testovania začína prebiehať validácia systému, teda overujeme nielen to, či modul pracuje korektne, ale aj to, či vykonáva správnu prácu. Testovanie je formálnejšie a viac kontrolované ako pri jednotkovom testovaní.

Môže sa zdať, že táto úroveň testovania je zbytočná, pretože ak všetky časti fungujú správne, mal by fungovať aj výsledný systém. Tento predpoklad je však nesprávny. Príkladom je systém, kde si dve časti predávajú medzi sebou dáta. Aj napriek tomu, že jednotlivé časti pracujú korektne, výsledný systém môže byť chybný, ak obe časti nepočítajú s rovnakou metrikou (napr. gramy a libry).

Integrácia je kombinovanie oddelených prvkov. Výsledná kombinácia by mala dokázať viac ako súčet jednotlivých prvkov. Integračné testovanie je vyhodnotenie kombinácie prvkov, či dosahujú požadované výsledky, keď sú použité spolu alebo súbežne. Toto je možné dosiahnuť tromi spôsobmi:

- testovanie balíčkov,
- testovanie spolupráce,
- testovanie funkčnosti.

Každý z týchto prístupov má svoj špecifický cieľ a každý môže hrať významnú rolu pri definícii testovacej sady aplikácie.

Testovanie balíčkov

Triedy v rámci Java aplikácie sú zoskupované do balíčkov. Testovanie na úrovni balíčkov [9] je typom jednotkového testovania, ktoré odzrkadľuje štruktúru balíčkov testovanej aplikácie. To znamená, že jednotkové testy môžu byť zoskupované do rovnakej štruktúry ako samotný zdrojový kód, čo môže významne pomôcť aj pri udržiavaní systému. Implementácia testovaných balíčkov predstavuje agregáciu všetkých tried v rámci balíčka do testovacej sady.

Testovanie spolupráce

Testovanie spolupráce [9] je postup integrácie jednotiek za účelom zhodnotenia, či jednotky spolupracujú. Logický je výber množiny takých tried a rozhraní, ktorá dokáže vyhodnotiť správanie medzi balíčkami. Toto testovanie je riadené vývojárom, kedy je potrebné otestovať funkčnosť vyžadovanú aplikáciou, ale ktorá nie je prísne vyžadovaná špecifikáciou produktu. Implementované je agregáciou logických množín prvkov (triedy a rozhrania) na dosiahnutie funkcie, ktorú je možné vnútorne otestovať v rámci aplikácie.

Testovanie funkčnosti

Na rozdiel od testovania spolupráce, testovanie funkčnosti [9] je riadené špecifikáciou alebo prípadmi použitia. Z hľadiska štruktúry sú testy rovnaké. Test riadený prípadmi použitia býva komplexnejší ako test spolupráce.

2.4.3 Systémové testovanie

Sytémové testovanie [9], známe ako end-to-end testovanie, je rozšírením integračného testovania. Cieľom je ohodnotiť všetky komponenty a/alebo spolupráce v rámci systému alebo aplikácie. Rozlišujeme dva typy:

- agregované testovanie balíčkov,
- agregované testovanie spolupráce.

Tieto dva typy sa nemusia nevyhnutne vylučovať. Oba majú svoje výhody a môžu byť použité spolu alebo súbežne.

Agregované testovanie balíčkov

Idea agregovaného testovania balíčkov [9] je rovnaká ako pri testovaní balíčkov v rámci integračného testovania. Každý balíček obsahuje testovacie prípady pre jednotkové testy v ňom zahrnuté. Každý balíček ďalej obsahuje testovaciu sadu zloženú zo všetkých testovacích prípadov, čo umožňuje ich sekvenčné spustenie.

Testy balíčkov môžu byť štruktúrované dvomi spôsobmi, a to včlenením testov priamo do balíčkov s testovanými triedami alebo ponechaním testy zvlášť v štruktúre balíčkov rovnakej ako samotná aplikácia. Výhoda integrovania testov so zdrojovými kódmi je, že cesta k testom je rovnaká ako cesta k triede, ktorú testujú.

Pre beh agregovaného balíčkového testu je potrebné vytvoriť hlavnú testovaciu sadu, ktorá zoskupuje a spúšťa všetky testovacie sady jednotlivých balíčkov.

Agregované testovanie spolupráce

Agregované testovanie spolupráce [9] aplikuje myšlienku testovania spolupráce a testovania funkčnosti z integračného testovania na celý systém. Existujú dva prístupy: agregácia všetkých testov spolupráce v rámci aplikácie alebo sada navrhnutá podľa funkčnej špecifikácie, ktorá pokrýva všetky prípady použitia spojené s aplikáciou. Zoskupenie testov spolupráce v rámci systému môže obsahovať všetky testy spolupráce a rovnako aj všetky testy funkčnosti.

Stratégie agregovaného testovania spolupráce môžu byť nasledovné:

- *Testovanie zhora nadol* – znamená testovanie systému od najviac abstraktnej úrovne po najviac konkrétnu. Táto stratégia je veľmi užitočná pri inkrementálnom vývoji softwaru.
- *Testovanie zdola nahor* – postupuje opačne ako testovanie zhora nadol, avšak štruktúra testov je rovnaká.
- *Úplné testovanie prípadov použitia* – sa dosiahne napísaním testov funkčnosti pre všetky prípady použitia definované v špecifikácii a ich spoločným spustením v hlavnej testovacej sade.
- *Testovanie kritických spoluprác* – začína analýzou kritických spoluprác v systéme, napr. spolupráca medzi business objektmi a prístupom k databáze. Potom je potrebné navrhnúť testy, ktoré vyhodnotia výhradne tieto spolupráce.

2.4.4 Databázové testovanie

Testovanie databázového kódu sa odlišuje od testovania ostatných druhov Java kódu, preto je potrebné venovať mu zvláštnu pozornosť. Databázové testovanie [9] má svoje komplikácie vynucujúce si zavedenie novej množiny testovacích vzorov.

Relačné databázy sú navrhnuté tak, aby mohli byť dotazované a aktualizované v paralelne bežiacich výpočetných vláknach. Viacvláknové testovanie je jednou z najdôležitejších častí procesu databázového testovania. Avšak najpoužívanejšie nástroje používané na testovanie Java kódu sú jednovláknové. Vo všeobecnosti existuje niekoľko druhov testov, ktoré skúmajú nasledujúce otázky vyplývajúce z viacvláknovej povahy databázového programovania:

- *Databázové zámky* – používajú sa na udržanie transakcií izolovaných od seba. Na otestovanie stratégie uzamykania je nutné spustiť testy paralelne. Je niekoľko spôsobov, ako sa to dá dosiahnuť, napr. spustenie viacerých kópií testov v rôznych JVM alebo spustenie viacerých vlákien v testoch.

- *Dlhodobé vzájomné pôsobenia* – vznikajú, keď metódy, ktoré sú v čase ďaleko od seba, môžu byť tesne prepojené.

- *Záťažové testovanie* (stress testing) – ďalším hľadiskom, ktoré treba vziať do úvahy pri používaní zámkov, je vplyv zamykania na výkon celého systému. Často sa jemné interakcie vyplývajúce z uzamykania prejavia, iba keď je systém pod záťažou.

Ako mnoho ďalších častí testovania, najlepším prístupom k testovaniu databázového kódu je začať s uvažovaním jednotlivých vrstiev aplikácie a následným vytvorením testov, ktoré nezávisle otestujú každú vrstvu.

2.5 Analýza pokrytia kódu

Jednotkové testovanie je rozhodujúcou časťou vývoja softwaru. Aj napriek tomu je často robené nedostatočne a nekvalitne. V zásade existujú dve príčiny, ktoré spôsobujú, že jednotkové testy sú neúčinné:

- nedostatočné otestovanie business logiky napriek vykonaniu kódu,

- zanedbanie niektorých častí kódu.

V prvom prípade je problém náročné detekovať automaticky. Druhú príčinu riešia nástroje na *analýzu pokrytia kódu* [17, 3] (code coverage).

Analýza pokrytia kódu je proces, ktorý zahŕňa vyhľadanie úsekov programu, ktoré nie sú pokryté množinou testovacích prípadov, vytvorenie dodatočných testovacích prípadov, ktoré zvýšia pokrytie a určenie úrovne pokrytia kódu, ktorá nepriamo určuje aj jeho kvalitu. Voliteľnou súčasťou môže byť aj identifikácia nadbytočných testovacích prípadov, ktoré nezvýšia pokrytie kódu. Pokrytie kódu testovacími prípadmi je teda užitočná metrika, ktorá vyjadruje, aké množstvo zdrojového kódu aplikácie je vykonané počas testov, inými slovami – ako dobre je daná aplikácia testovaná.

Pri analýze pokrytia kódu sa využívajú viaceré kritériá, medzi základné patria:

- *Pokrytie príkazov* [3] (statement coverage) – informuje o tom, či každý vykonateľný príkaz bol vykonaný. Vykonateľný príkaz je taký deklaratívny príkaz, ktorý generuje vykonateľný kód. Nevýhodou tejto metriky je, že nedokáže rozlíšiť niektoré riadiace štruktúry, koncovú podmienku cyklov, po sebe nasledujúce návestia príkazu switch a logické operátory. Jedinou výhodou oproti ostatným kritériám je tá, že podiel pokrytých príkazov odráža podiel odhalených chýb, pretože programové chyby bývajú rovnomerne rozptýlené v zdrojovom kóde.

- *Pokrytie základných blokov* [3] (basic block coverage) – je podobné pokrytiu príkazov, avšak jednotkou meraného kódu nie je príkaz, ale základný blok. Základný blok je taká postupnosť inštrukcií bajtkódu, ktorá neobsahuje skok alebo cieľ skoku. To znamená, že ak je vykonaná jedna inštrukcia základého bloku, tak budú vykonané aj všetky ostatné inštrukcie tohto základného bloku. Kvôli možnosti vzniku výnimky uprostred základného bloku sa väčšinou základný blok považuje za pokrytý, ak je vykonaná

jeho posledná inštrukcia. V takom prípade stopercentné pokrytie základných blokov implikuje stopercentné pokrytie príkazov.

- *Pokrytie riadkov* [3] (line coverage) – vyjadruje, koľko percent riadkov bolo vykonaných. Závisí od formátovania zdrojového kódu, jeden riadok môže obsahovať jeden alebo viac základných blokov a jeden základný blok môže byť rozdelený do niekoľkých riadkov.

- *Pokrytie rozhodnutí* [3] (decision coverage, branch coverage) – sleduje, či všetky boolovské výrazy testované v riadiacich štruktúrach boli vyhodnotené aj ako pravdivé, aj ako nepravdivé. Táto metrika je jednoduchá a zároveň nemá problémy, ktoré sa vyskytujú pri pokrytí príkazov. Nevýhodou je, že ignoruje vetvenie programu v rámci boolovského výrazu, ktoré vzniká v dôsledku skratového vyhodnotenia logického výrazu.

- *Pokrytie podmienok* [3] (condition coverage) – si všíma pozitívny alebo negatívny výsledok každej podmienky, zohľadňuje podmienky nezávisle na sebe. Toto kritérium je podobné ako pokrytie rozhodnutí, ale lepšie rozlišuje riadiace toky.

- *Pokrytie zložených podmienok* [3] (multiple condition coverage) – sleduje, či sa vyskytnú všetky kombinácie podmienok, ktoré prichádzajú do úvahy. V jazykoch so skratovým vyhodnotením logických výrazov (napr. Java, C/C++) je pokrytie zložených podmienok podobné pokrytiu podmienok a jeho výhodou je, že vyžaduje veľmi dôkladné testovanie. Nevýhodou je, že môže byť náročné nájsť minimálnu množinu požadovaných testovacích prípadov, hlavne pri zložitých boolovských výrazoch.

- *Pokrytie podmienok a rozhodnutí* [3] (condition/decision coverage) – je hybridným kritériom, ktoré vzniklo spojením pokrytia podmienok a pokrytia rozhodnutí. Jeho výhodou je jednoduchosť a neprítomnosť nevýhod jednotlivých zložiek.

- *Pokrytie ciest* [3] (path coverage) – sleduje, či sa program riadil každou z možných ciest v každej metóde. Cesta je špecifická sekvencia vetiev v metóde od vstupu až po výstup. Podobne ako porytie zložených podmienok, aj pokrytie ciest vyžaduje veľmi dôkladné testovanie. Nevýhodou tejto metriky je veľké množstvo ciest, ktoré exponenciálne rastie s množstvom vetiev programu.

- *Pokrytie metód* [3] (method coverage) – určuje, koľko metód bolo počas testov zavolaných.

Odlišnou metrikou analýzy pokrytia kódu je *pokrytie synchronizácie* [2] (synchronization coverage), ktoré sa na rozdiel od predchádzajúcich kritérií používa pri testovaní viacvláknových aplikácií. Zameriava sa na testovanie synchronizačných príkazov, ktoré zabezpečujú výlučný prístup jedného vlákna k chránenému kódu. Vo väčšine prípadov nedôjde počas vykonávaniu chráneného kódu k prepnutiu kontextu, pretože takéto bloky kódu bývajú čo najkratšie. Aby sa prejavil vplyv synchronizácie, je potrebný test, v ktorom sa preruší vykonávanie vlákna alebo ktorý spôsobí prerušenie iného vlákna. Jednoduchý spôsob, ako ukázať, že synchronizačné príkazy nie sú vôbec testované, je ich odstránenie a zistenie, že testy naďalej uspejú. Na rozdiel od ostatných metrík, pokrytie synchronizácie neposkytuje deterministické výsledky, a preto sa pri ňom testy spúšťajú opakovane.

Projekty majú zvyčajne dopredu určené hodnoty pokrytia kódu testami, ktoré musia dodržať pre jednotlivé kritériá. Prirodzene, kritické systémy majú tieto hranice vyššie ako bežné aplikácie. Pokrytie 100 % však mnohokrát nie je možné dosiahnuť.

Nástroje na analýzu pokrytia kódu pracujú pomocou tzv. *inštrumentácie*, kedy sa do aplikácie vloží kód, ktorý zabezpečí zbieranie dát o pokrytí. Takýmto kódom je napríklad počítadlo, ktoré sa zvyšuje pri každom prechode skúmaným úsekom zdrojového kódu. Inštrumentáciu je možné prevádzať buď *staticky* – prebieha pred nahraním tried do JVM a inštrumentované triedy sú uložené na disk, alebo *dynamicky* – prebieha v JVM počas nahrávania tried. Iné rozdelenie je na inštrumentáciu zdrojového kódu, inštrumentáciu Java bajtkódu alebo použitie modifikovaného JVM.

Následne po inštrumentácii je potrebné spustiť testy aplikácie, počas ktorých sa zbierajú dáta pre výsledné štatistiky. Pre analýzu pokrytia kódu nie je vždy nevyhnutné mať automatizovanú sadu testov, pri niektorých nástrojoch je možné aplikáciu testovať aj manuálne. Z nazbieraných dát sú potom vygenerované správy o pokrytí v rôznych formátoch v závislosti od použitého nástroja.

2.5.1 Vybrané nástroje

V tejto podkapitole budú predsavené niektoré nástroje na analýzu pokrytia kódu, ktoré sú k dispozícii pre Java projekty. Tieto nástroje sa od seba líšia v spôsobe inštrumentácie, meraných metrikách pokrytia kódu, formáte výslednej správy o pokrytí aj spôsobe použitia.

Emma

Emma[3][16] je open source nástroj na meranie a analýzu pokrytia kódu testami, ktorý podporuje statickú aj dynamickú inštrumentáciu bajtkódu. Jeho funkcie je možné využiť z príkazového riadku alebo integrovať do zostavovacieho procesu pomocou aplikácie Ant [8].

Na vyhodnotenie pokrytia kódu Emma uchováva dva druhy dát v rôznych súboroch. Prvým sú metadáta ukladané v súboroch s príponou .em, ktoré obsahujú statické dáta. Statické dáta zahŕňajú informácie o jednotlivých triedach, metódach, riadkoch či blokoch ako napr. počet, umiestnenie v konkrétnych zdrojových súboroch apod. Druhým typom sú dynamické dáta o pokrytí, získavané počas behu testov, ktoré sa ukladajú do súborov s príponou .ec. Tieto dáta osahujú napr. informácie, koľkokrát bol daný základný blok vykonaný alebo či bola zavolaná daná metóda. Posledným typom sú kombinované dáta v .es súboroch, ktoré obsahujú metadáta i dáta o pokrytí. Pre generovanie správy o pokrytí kódu je možné použiť rôznu kombináciu týchto súborov v ľubovoľnom množstve, prípadne dáta z viacerých súborov spojiť do jedného.

Emma vyhodnocuje štatistiky na úrovni celého projektu, ale aj na úrovni jednotlivých balíčkov, tried, či metód. Správa o pokrytí môže mať formát textu, HTML alebo XML. Do HTML správy je možné pripojiť zdrojové kódy alebo zvýrazniť prvky s pokrytím menším ako určitá užívateľom zadaná hranica.

Cobertura

Cobertura[4] [17] je open source nástroj, ktorý je možné použiť z príkazového riadku, ale aj integrovať s nástrojom Ant alebo Maven [8].

Okrem klasického pokrytia príkazov a pokrytia rozhodnutí Cobertura meria aj priemernú *cyklomatickú zložitosť* [17] (angl. cyclomatic complexity) tried a balíčkov. Cyklomatická zložitosť nie je kritérium pre analýzu pokrytia kódu, ale patrí k užitočným metrikám, ktoré

[3] http://emma.sourceforge.net/
[4] http://cobertura.sourceforge.net/

vyjadrujú zložitosť zdrojového kódu. Dlhé a zložité úseky kódu sú náchylné k chybám a ťažšie sa udržiavajú, preto je vhodné ich automaticky detekovať a následne refaktorovať.

Informácie o inštrumentovaných triedach sú postupne ukladané do súboru cobertura.ser, z ktorého je nakoniec vygenerovaná správa o pokrytí. Správa môže mať formát HTML alebo XML. Implicitným nastavením je formát HTML, ktorý poskytuje prehľadné a intuitívne zobraznie informácií o jednotlivých triedach, balíčkoch a celom projekte. Odkaz do zdrojového kódu umožní jednoduché identifikovanie netestovaných riadkov, ale aj nadbytočne testované riadky.

Prostredníctvom Cobertury je možné vynútiť si minimálne pokrytie kódu priamo v zostavovacom procese pomocou vloženia kontroly pokrytia do zostavovacieho skriptu. Zostavenie projektu skončí neúspechom, pokiaľ je hodnota pokrytia nižšia ako zadaná minimálna hranica. Táto hranica môže byť nastavená pre celý projekt, vybrané balíčky alebo jednotlivé triedy zvlášť pre pokrytie príkazov a pokrytie rozhodnutí.

Clover

Clover[5] [11] je komerčný software na meranie pokrytia kódu. Na rozdiel od predchádzajúcich používa inštrumentáciu zdrojového kódu, ktorá sa vykonáva pred kompiláciou projektu. Informácie o štruktúre projektu a jednotlivých súboroch sú počas inštrumentácie uložené do databázy coverage.db. Clover je možné integrovať s nástrojmi Ant a Maven alebo využiť jeho nátroje príkazového riadku či zásuvné moduly pre rôzne vývojové prostredia.

Clover poskytuje meranie pokrytia príkazov, rozhodnutí a metód. Tieto hodnoty potom používa na výpočet celkového pokrytia podľa vzorca (2.1), kde BT je počet vetiev, ktoré boli vyhodnotené počas testov aspoň raz ako pravdivé, BF počet vetiev vyhodnotených ako nepravdivé, SC počet vykonaných príkazov, MC počet zavolaných metód, B počet vetiev, S počet príkazov a M počet metód.

$$TPC = \frac{BT + BF + SC + MC}{2B + S + M} \tag{2.1}$$

Tento nástroj oproti ostatným ponúka optimalizáciu testov, ktorá môže urýchliť proces kompilácie a behu testov. Clover zaznamenáva konkrétne testy, ktoré pokrývajú jednotlivé riadky zdrojového kódu. Vďaka tejto vlastnosti dokáže určiť a spustiť len testy pokrývajúce kód, ktorý prešiel nejakou zmenou od posledného behu testov. Druhou možnosťou je spustiť všetky testy, ale vo vhodnom poradí – najprv testy v poslednom behu neúspešné, následne testy pokrývajúce kód so zmenami a nakoniec ostatné testy.

CodePro Analytix

CodePro Analytix[6] [12] je nástroj na testovanie Java aplikácií pre vývojové prostredie Eclipse[7]. Okrem merania pokrytia kódu zahŕňa aj iné užitočné funkcie: analýza zdrojového kódu, analýza závislostí, analýza podobného kódu, generátor a editor JUnit testov, hodnotenie kvality zdrojového kódu.

Analýza pokrytia kódu vykonávaná týmto nástrojom je založená na mechanizme nástroja Emma. Z metrík pokrytia kódu sú k dispozícii pokrytie tried, metód, riadkov, základných blokov a príkazov. CodePro Analytix poskytuje statickú i dynamickú inštrumentáciu tried.

[5] http://confluence.atlassian.com/display/CLOVER/Clover+Documentation+Home
[6] http://code.google.com/intl/cs/javadevtools/codepro/doc/index.html
[7] http://www.eclipse.org/

Po inštrumentácii sa dáta o pokrytí zbierajú pri každom spustení aplikácie. Pokiaľ nechceme tieto dáta zbierať, napr. v prípade, že aplikácia bude distribuovaná na ďalšie počítače, je nutné triedy odinštrumentovať.

Výsledky merania pokrytia kódu sú zaznamenávané po každom spustení aplikácie a ukladané na disku. Vďaka tomu je možné zobraziť históriu pokrytia projektu a vidieť, ako sa pokrytie kódu vyvíjalo. Okrem náhľadu na výsledky priamo v Eclipse CodePro Analytix generuje aj správu o pokrytí vo formáte textového dokumentu, HTML alebo XML. Do týchto správ je rovnako možné zahrnúť aj históriu pokrytia.

ConTest

ConTest [15] patrí ku komerčným nástrojom a slúži na hľadanie chýb a analýzu pokrytia kódu viacvláknových aplikácií. Je možné ho použiť ako samostatnú aplikáciu alebo ako zásuvný modul pre vývojové prostredie Eclipse. ConTest poskytuje statickú aj dynamickú inštrumentáciu bajtkódu, pri ktorej vkladá na vybrané miesta volanie svojich funkcií. Sú to také miesta, pri ktorých je pravdepodobné, že ich relatívne poradie v rámci vlákna zmení výsledok, napr. vstup a výstup z chráneného bloku, prístup k zdieľaným premenným apod. Implementované metriky pokrytia kódu zahŕňajú pokrytie metód, základných blokov a pokrytie synchronizácie.

Pre jednovláknové aplikácie je vhodné použiť tento nástroj na meranie pokrytia metód a základných blokov. V tom to prípade po spustení testov vytvorný adresár s *.trace* súbormi, ktoré obsahujú informácie o pokrytí. ConTest poskytuje nastavenie, s ktorým vynechá meranie pokrytia takých častí kódu, ktoré boli pokryté v predchádzajúcich behoch testov. Takto je možné docieliť rýchlejšie spustenie testovacej sady po dopísaní nových testov. ConTest neposkytuje generovanie správ o pokrytí. Na tento účel slúži aplikácia FoCuS[8].

Pokrytie synchronizácie je metrika, ktorá pri predchádzajúcich nástrojoch chýba. ConTest v súčasnosti meria pokrytie synchronizačných primitív `synchronized` a `Object.wait`, do budúcnosti sa počíta s ďalšími ako `notify`, `join` a `interrupt`. Každý synchronizačný blok poskytuje tri typy udalostí. Synchronizácia môže byť teda *zasiahnutá* (synchronization visited), *blokujúca* (synchronization blocking) alebo *blokovaná* (synchronization blocked). Navyše primitívum `Object.wait` posyktuje ďalšie dve – *čakanie* (wait visited) a *opakované čakanie* (wait repeated).

K pokrytiu synchronizácie navyše patrí *pokrytie zdieľaných premenných* (shared variables coverage) a *pokrytie párov súvesejúcich príkazov* (interfered location pairs coverage). Výsledkom merania pokrytia zdieľaných premenných je trace súbor so zoznamom premenných, ktoré boli použité vo viac ako jednom vlákne.

[8] http://www.alphaworks.ibm.com/tech/focus

Kapitola 3

Analýza pokrytia kódu JBoss AS

V tejto kapitole popíšem detailný postup použitia dvoch nástrojov na analýzu pokrytia kódu testovacou sadou aplikačného servera JBoss. Hlavnými kritériami pre výber použitých nástrojov boli schopnosť pracovať s Java aplikáciami, keďže majú byť použité na aplikačný server JBoss, voľná dostupnosť a statická inštrumentácia, pretože s Java EE aplikačným serverom nie je možné použiť dynamickú inštrumentáciu[1]. JBoss AS používa na zostavenie skripty aplikácie Ant, ďalšou požadovanou vlastnosťou nástrojov je teda možnosť integrovania do zostavovacieho procesu pomocou tejto aplikácie. Vhodné by bolo aj, aby vybrané nástroje ponúkali širokú škálu metrík pokrytia, na druhú stranu by ale mali mať aspoň jednu metriku spoločnú, aby ich bolo možné porovnať. Z nástrojov som nakoniec vybrala Coberturu a Emmu, z verzií aplikačného servera JBoss AS 3.2 až po JBoss AS 5.1.

Celý postup som rozdelila do troch krokov:

1. príprava na zbieranie dát,
2. spustenie testovacej sady,
3. samotná analýza nazbieraných dát.

Tieto kroky rozoberiem v nasledujúcich podkapitolách, najprv všeobecne a potom konkrétne pre každý nástroj. V podkapitole 3.3.2 uvediem prehľad výsledkov, ktoré som získala použitím jednotlivých nástrojov. Úplné správy o pokrytí sa nachádzajú na priloženom CD.

Na inštrumentáciu a generovanie správy o pokrytí som vytvorila skript aplikácie Ant umiestnený v adresári *code-coverage/* s názvom *code-coverage.xml*. Tento skript obsahuje spustiteľné ciele `emma-instrument` a `cobertura-instrument`, príp. `emma-report` a `cobertura-report`.

3.1 Príprava na zbieranie dát

Na meranie pokrytia kódu potrebujeme zdrojové súbory aplikačného servera JBoss, ktoré sú voľne stiahnuteľné z oficiálnych stránok JBoss[2]. Ku kompilácii aplikačného servera je potrebné mať nainštalovanú správnu verziu JDK a aplikácie Ant[3] (Tabuľka 3.1) a nastavené systémové premenné `JAVA_HOME` a `ANT_HOME`.

[1] http://emma.sourceforge.net/faq.html#q.runtime.appservers
[2] http://www.jboss.org/jbossas/downloads
[3] http://archive.apache.org/dist/ant/binaries/

verzia AS	verzia JDK	verzia Ant
AS 3	1.4	1.6
AS 4	1.5	1.6
AS 5	1.6	1.7

Tabuľka 3.1: Verzie JDK a Ant použité pre príslušné verzie JBoss AS.

Kompilácia sa prevedie pomocou skriptu *build.xml* v adresári *build/*:

```
ant -buildfile build/build.xml
```

Aby bolo možné získať informácie o jednotlivých riadkoch zdrojových kódov, je nutné kompilovať aplikačný server s ladiacimi informáciami:

```
ant -buildfile build/build.xml -Djavac.debug=true
    -Djavac.debuglevel=lines,source
```

Ďalší postup prípravy zahŕňa inštrumentáciu a prípadne ďalšiu konfiguráciu v závislosti od použitého nástroja, a preto bude popísaný zvlášť.

3.1.1 Emma

Pre použitie nástroja Emma je potrebné mať súbory *emma.jar* a *emma_ant.jar*[4] viditeľné pre aplikáciu Ant zo súborov *code-coverage.xml* a *testsuite/build.xml*, čo dosiahneme vložením riadkov na Obrázku 3.1 na ich začiatok.

```
<property name="emma.dir" value="cesta_k_emme" />
    <path id="emma.lib">
        <fileset dir="${emma.dir}">
            <include name="emma.jar"/>
            <include name="emma_ant.jar"/>
        </fileset>
    </path>
<taskdef resource="emma_ant.properties" classpathref="emma.lib" />
```

Obrázok 3.1: Definícia umiestnenia súborov Emmy.

Inštrumentovanie aplikačného servera pomocou nástroja Emma sa spustí príkazom:

```
ant emma-instrument -f code-coverage/code-coverage.xml
```

Cieľ `emma-instrument` obsahuje úlohu `instr` s nasledujúcimi atribútmi:

- `instrpathref="instrument.path"` – cesta k súborom typu .class a .jar, ktoré majú byť inštrumentované.
- `metadatafile="coverage.em"` – súbor s metadátami sa vytvorí v koreňovom adresári aplikačného servera.

[4] http://sourceforge.net/projects/emma/files/emma-release/

- `mode="overwrite"` – pôvodné súbory budú prepísané inštrumentovanou verziou kvôli tomu, aby testy pracovali s inštrumentovanými triedami.
- `merge="true"` – metadáta budú zlúčené do jedného súboru.
- `filter="+org.jboss.*,+org.jgroups.*,+org.hibernate.*,+com.arjuna.*, -*.tests.*, -*.samples.*"` – filter, podľa ktorého prebehne inštrumentácia tried. Z tried, ktoré sa nachádzajú na zadanej ceste budú inštrumentované len tie, ktoré nás zaujímajú (označené znamienkom „+“), všetky ostatné ostanú nezmenené. Nezmenené ostanú aj triedy označené znamienkom „-“, aj keď zároveň vyhovujú prvej časti filtra. V tomto prípade sú to triedy, ktoré patria k testom alebo príkladom. Takéto triedy sa logicky nezahŕňajú do merania pokrytia kódu.

Pri takto inštrumentovanom aplikačnom serveri Emma neposkytuje meranie pokrytia príkazov. Je to z toho dôvodu, že medzi inštrumentovanými triedami sa nachádzajú aj také, ktoré sú súčasťou knižníc, a teda nemáme k dispozícii ich zdrojové kódy a pravdepodobne neboli kompilované s ladiacimi informáciami. Na to, že inštrumentujeme aj takéto triedy nás Emma upozorní vo výpise počas inštrumentácie:

```
[instr] package [org/jboss/axis/wsdl/fromJava] contains
        classes [Emitter] without full debug info
```

Pokiaľ chceme, aby v správe o pokrytí vystupovali aj informácie o pokrytí príkazov, je nutné filter rozšíriť a vyradiť z procesu inštrumentácie nevyhovujúce triedy. Pre predchádzajúcu triedu doplníme vyššie uvedený filter o časť `-org.jboss.axis.wsdl.fromJava.Emitter`. Nakoniec je potrebné ešte skopírovať súbor *emma.jar* do adresára *cesta_k_jave/jre/lib/ext/* a aplikačný server je pripravený na spustenie testovacej sady pre zber dát.

3.1.2 Cobertura

Jednou z knižníc, ktoré Cobertura[5] obsahuje je log4j.jar. Pred jej použitím sa musíme presvedčiť, že máme správnu verziu, pretože pre úspešné meranie pokrytia kódu potrebujeme verziu 1.2.12 alebo novšiu namiesto tej, ktorá je dodávaná s aktuálnou verziou Cobertury. Vo svojej práci som použila log4j-1.2.16.jar[6]. Cobertura musí byť tiež viditeľná pre aplikáciu Ant. Toto docielime podobnou úpravou súborov *code-coverage.xml* a *testsuite/build.xml*, ako v prípade použitia Emmy (Obrázok 3.2).

Inštrumentovanie s Coberturou sa spúšťa príkazom:

```
ant cobertura-instrument -f code-coverage/code-coverage.xml
```

Najdôležitejšou úlohou volaného cieľa je `cobertura-instrument` (Obrázok 3.3). V tomto prípade bude Cobertura inštrumentovať triedy, ktoré sa nachádzajú na ceste `instrument.path`. Vnorené úlohy `includeClasses` a `excludeClasses` majú rovnaký význam ako filter v Emme. V oboch je atribútom regulárny výraz, ktorý predstavuje triedy, ktoré majú byť inštrumentované (includeClasses) a ktoré majú byť z inštrumentácie vylúčené (excludeClasses).

Cobertura nevyžaduje ďalšie rozšírenie filtra o triedy, pre ktoré nemáme zdrojové súbory alebo ladiace informácie z prekladu. Pre tieto triedy jednoducho nebude pokrytie merané.

[5] http://cobertura.sourceforge.net/download.html
[6] http://logging.apache.org/log4j/1.2/download.html

```
<property name="cobertura.dir" value="cesta_ku_Coberture" />
<path id="cobertura.classpath">
    <fileset dir="${cobertura.dir}">
        <include name="cobertura.jar" />
        <include name="lib/**/*.jar" />
    </fileset>
</path>
<taskdef classpathref="cobertura.classpath"
         resource="tasks.properties" />
```

Obrázok 3.2: Definícia umiestnenia súborov Cobertury.

```
<cobertura-instrument>
    <includeClasses regex="org.\jboss\..*" />
    <includeClasses regex="org.\jgroups\..*" />
    <includeClasses regex="org.\hibernate\..*" />
    <includeClasses regex="com.\arjuna\..*" />
    <excludeClasses regex=".*\.tests\..*" />
    <excludeClasses regex=".*\.samples\..*" />
    <instrumentationClasspath>
        <path refid="instrument.path" />
    </instrumentationClasspath>
</cobertura-instrument>
```

Obrázok 3.3: Inštrumentovanie pomocou nástroja Cobertura.

Posledným krokom pred spustením testov je nastavenie časového limitu pre spustenie servera počas testovania, pretože Cobertura výrazne spomaľuje jeho beh. Implicitný limit je nastavený na 120 sekúnd, na 300 sekúnd ho zmeníme príkazom:

```
export ANT_OPTS="-Djbossas.startup.timeout=300
```

3.2 Spustenie testovacej sady

Testovacia sada JBoss AS je súčasťou zdrojových súborov každej verzie. Na zostavenie a spustenie testov a generovanie správy s výsledkami testov je pripravený skript v adresári *testsuite/*, ktorý sa spustí nasledovne:

```
sh testsuite/build.sh tests
```

Väčšina testov používa jednu inštanciu aplikačného servera. Tá je spúšťaná na IP adrese 127.0.0.1, teda localhost. Niektoré testy však pre svoj beh potrebujú dve inštancie aplikačného servera spustené na rôznych adresách. Pre druhú inštanciu servera sa používa IP adresa hostiteľského počítača, ktorú je potrebné nastaviť. Jedným spôsobom je vytvorenie súboru *local.properties* v adresári *testsuite/* s obsahom:

```
node0=127.0.0.1
node1=moje_pc
```

Druhou možnosťou je predať parametre node0 a node1 pri spustení testovacej sady:

```
sh testsuite/build.sh tests -Dnode0=127.0.0.1 -Dnode1=moje_pc
```

V oboch prípadoch `moje_pc` predstavuje IP adresu hostiteľského počítača.

Testy sú rozdelené do skupín, ktoré je možné volať aj samostatne. Pre každú skupinu testov je v skripte build.xml jeden cieľ, ktorých počet je v závislosti od verzie 8 - 26. Prehľad cieľov verzie 5.1.0 je v Tabuľke 3.6.

3.3 Analýza nazbieraných dát

Postupom uvedeným v podkapitolách 3.1 a 3.2 som získala dáta potrebné pre analýzu pokrytia kódu aplikačného servera JBoss. Nástroj Emma je možné použiť na všetky jeho verzie, pretože funguje od verzie Java 1.2. Aktuálna verzia Cobertury však používa verziu Java 1.5, preto som Coberturu použila len na tri novšie verzie aplikačného servera. Bolo by možné použitie staršej verzie Cobertury, ale výsledky by nemuseli byť porovnateľné kvôli rozdielnej implementácii. Na porovnanie dvoch nástrojov poslúžia aj tieto namerané dáta.

3.3.1 Generovanie správy o pokrytí

Počas behu testov získame obrovské množstvo dát, ktoré je vhodné spracovať tak, aby mali určitú vypovedaciu hodnotu. Preto nástroje na analýzu pokrytia kódu väčšinou poskytujú možnosť generovania správy o pokrytí vo vhodnom formáte. V súbore *code-coverage.xml* sú na tento účel ciele `emma-report` a `cobertura-report`.

Emma

Cieľ `emma-report` vytvorí správu vo formáte HTML v adresári *code-coverage/* (Obrázok 3.4). Vnorený element `sourcepath` určuje cestu k zdrojovým súborom aplikačného

```
<report depth="method">
    <sourcepath>
        <dirset dir="${basedir}">
            <include name="**/src/main"/>
        </dirset>
    </sourcepath>
    <infileset dir="${basedir}" includes="*.em, *.ec" />
    <html outfile="${basedir}/code-coverage/coverage.html" />
</report>
```

Obrázok 3.4: Generovanie správy o pokrytí nástrojom Emma.

servera. Súbory s dátami o pokrytí sa počas behu testov automaticky vytvorili v adresároch *testsuite/*, *testsuite/output/* a *build/output/verzia_AS/bin/* príslušnej verzie aplikačného servera.

Takto vygenerovné správy o pokrytí sa nachádzajú na priloženom CD. Súhrn informácií z týchto správ je zobrazený v Tabuľke 3.1.

Cobertura

V prípade Cobertury je vhodné vygenerovať správu o pokrytí aj vo formáte XML, pretože oproti formátu HTML poskytuje navyše informácie o pokrytí riadkov na úrovni jednotlivých metód. Keďže Cobertura podobne ako Emma vytvorí niekoľko súborov s dátami o pokrytí, ale správu dokáže vygenerovať len z jediného súboru, je potrebné najprv všetky súbory zlúčiť do jedného a až potom vygenerovať správu. Cieľ `cobertura-report` teda obsahuje dve hlavné úlohy:

```
<cobertura-merge maxmemory="1024M">
    <fileset dir=".">
        <include name="testsuite/cobertura.ser" />
        <include name="testsuite/output/cobertura.ser" />
        <include name="build/output/verzia_AS/bin/cobertura.ser"/>
    </fileset>
</cobertura-merge>

<cobertura-report format="xml"
    destdir="${basedir}/code-coverage/coverage-report"
    srcdir="${basedir}/code-coverage/coverage-report/src"
    datafile="cobertura.ser">
</cobertura-report>
```

Obrázok 3.5: Generovanie správy o pokrytí nástrojom Cobertura.

3.3.2 Vývoj pokrytia kódu testami rôznych verzií JBoss AS

Cieľom použitia viacerých verzií aplikačného servera bolo zistiť, ako sa postupne vyvíjalo pokrytie kódu jeho testovacou sadou.

Informácie získané nástrojom Emma majú tri časti – s inštrumentovaným kompletným aplikačným serverom (v Tabuľke 3.2 označené ako *a* pre príslušnú verziu), inštrumentované len triedy, ktoré majú dostupnný aj zdrojový kód a boli preložené s ladiacimi informáciami (*b*) a aplikačný server bez samostatných komponentov (*c*). Výsledky získané pomocou nástroja Cobertura sú v Tabuľke 3.3. Verzia 3.2.8 neobsahuje samostatné komponenty, preto časť (*c*) chýba. Naopak verzia 5.1.0 má štvrtú časť, v ktorej sú z inštrumentácie vynechané triedy s pokrytím tried 0%.

Z Tabuľky 3.2 a Tabuľky 3.3 vyplýva, že sa projekt postupne zväčšoval a pokrytie kódu mierne rástlo pre všetky merané kritériá. Pri bližšom pohľade je vidieť, že hodnoty namerané rôznymi nástrojmi sa pre spoločnú metriku pokrytie riadkov nezhodujú. Táto odlišnosť bude diskutovaná v podkapitole 3.3.3.

Správy o pokrytí obsahujú podrobné informácie o jednotlivých balíčkoch či triedach. Ako prehľad uvádzam v Tabuľke 3.4 jednoduchú štatistiku počtu tried jednotlivých verzií JBoss AS zoskupených podľa hodnoty pokrytia základných blokov. Triedy, ktoré nie sú testované, majú túto hodnotu menšiu ako 20%. Pri testovaných triedach vyjadruje hodnota menšia ako 50% slabé pokrytie, hodnota v intervale 50-80% nedostatnočné pokrytie a hodnota nad 80% dobré pokrytie.

Verzia	pokr. tried	pokr. metód	pokr. blokov	pokr. riadkov	poč. tried
3.2.8 a	51%	36%	32%	-	3761
4.0.5 a	60%	34%	30%	-	6311
4.2.3 a	51%	36%	32%	-	8915
5.0.1 a	65%	41%	36%	-	12945
5.1.0 a	63%	40%	36%	-	14621
3.2.8 b	51%	38%	33%	34%	3075
4.0.5 b	59%	34%	30%	32%	6216
4.2.3 b	58%	33%	29%	31%	8817
5.0.1 b	65%	41%	36%	39%	12688
5.1.0 b	63%	40%	36%	38%	14331
4.0.5 c	62%	38%	32%	35%	4307
4.2.3 c	62%	39%	34%	36%	5020
5.0.1 c	66%	44%	38%	41%	8380
5.1.0 c	64%	43%	38%	41%	9148
5.1.0 d	100%	61%	51%	54%	7722

Tabuľka 3.2: Výsledky namerané nástrojom Emma.

Verzia	pokrytie riadkov	pokrytie rozhodnutí
4.2.3	31%	24%
5.0.1	35%	27%
5.1.0	36%	28%

Tabuľka 3.3: Výsledky namerané nástrojom Cobertura.

JBoss AS 3.2.8

Najstaršia z analyzovaných verzií aplikačného servera JBoss je počtom balíčkov najmenšia a podľa výsledkov merania pokrytia kódu má najväčší podiel netestovaných tried (Tabuľka 3.2 a Tabuľka 3.3). Z 202 balíčkov až 64 nebolo testami vôbec dotknutých. V tomto období sa totiž testovacia sada len začala vyvíjať.

Najviac tried patrí k netestovaným. Z testovaných tried má polovica nedostatočné pokrytie. Dôkladne testovaných tried s dobrým pokrytím základných blokov je 452 (Tabuľka 3.4). Táto hodnota predstavuje necelých 15% z celkového počtu tried.

JBoss AS 4.0.5

V tejto verzii oproti predchádzajúcej pribudli samostatné komponenty, ktoré majú vlastnú testovaciu sadu mimo testovacej sady JBoss AS. Tieto komponenty nie sú teda testované, môžu však byť v niektorých testoch potrebné. To viedlo k miernemu poklesu pokrytia (Tabuľka 3.2 a Tabuľka 3.3). K týmto komponentom patria: `org.hibernate`, `org.jboss.aspects`, `org.jboss.remoting` a `org.jboss.ws`.

Rovnako klesol aj podiel tried s dobrým pokrytím o 1% a naopak pomer netestovaných tried stúpol o 3% (Tabuľka 3.4).

pokrytie blokov	AS 3.2.8	AS 4.0.5	AS 4.2.3	AS 5.0.1	AS 5.1.0
< 20	1544	3283	4653	5810	6926
< 50	352	896	1116	1928	2096
< 80	727	1162	1803	2833	3077
≥ 80	452	875	1245	2117	2232

Tabuľka 3.4: Počty tried v jednotlivých verziách podľa pokrytia blokov.

JBoss AS 4.2.3

Vo verzii 4.2.3 pribudli ďalšie samostatné komponenty `com.arjuna`, `org.jboss.aop` a `org.jboss.wsf`. To a ďalší nárast projektu viedli k dodatočnému poklesu pokrytia aj napriek tomu, že testovacia sada sa neustále rozširovala (Tabuľka 3.2 a Tabuľka 3.3).

JBoss AS 5.0.1

Táto verzia dosiahla najvyššie hodnoty pokrytia pre všetky merané metriky. Rovnako tu nastal najväčší nárast počtu tried (Tabuľka 3.2 a Tabuľka 3.3). Zo samostatných komponentov pribudol len jeden, `org.jboss.seam`.

Najvyššiu hodnotu dosiahol aj pomer dobre pokrytých tried z celkového počtu tried, a to 17% (Tabuľka 3.4).

JBoss AS 5.1.0

V tejto fáze dosiahol aplikačný server takmer 4-násobok veľkosti oproti prvej verzii. Testovacia sada sa postupne vyvíjala, napriek tomu pokrytie kódu mierne kleslo (Tabuľka 3.2 a Tabuľka 3.3). Opäť pribudol jeden samostatný komponent – `org.jboss.webbeans`.

Triedy, ktoré majú pokrytie tried 0%, neboli počas behu testov vôbec použité. Vynechaním týchto tried z inštrumentácie som získala hodnoty, ktoré sa približujú tým, ktoré by som získala dynamickou inštrumentáciou (Tabuľka 3.2 – riadok AS 5.1.0 *d*). Ako sa dalo čakať, pokrytie tried je v tomto prípade 100% a aj všetky ostatné metriky vykazujú výrazne vyššie hodnoty.

3.3.3 Porovnanie výsledkov od rôznych nástrojov

Oba nástroje, Emma aj Cobertura majú spoločnú len jednu metriku – pokrytie riadkov. Avšak aj pri tejto jednej metrike je vidieť, že sa ich hodnoty líšia.

verzia	Emma		Cobertura	
	pokrytie riadkov	počet tried	pokrytie riadkov	počet tried
AS 4.2.3	31%	8817	31%	10611
AS 5.0.1	39%	12088	35%	15081
AS 5.1.0	38%	14331	36%	16653

Tabuľka 3.5: Porovnanie výsledkov nameraných rôznymi nástrojmi.

Odlišnosť vo výsledkoch súvisí s rozdielnym počtom tried, ktoré jednotlivé nástroje inštrumentujú, a teda z nich zbierajú dáta pre meranie pokrytia. Emma na rozdiel od

Cobertury vynecháva nasledujúce položky:

- *anonymné triedy* – pretože ich Emma pravdepodobne nevie rozlíšiť,
- *abstraktné rozhrania* – pretože nie sú potrebné pre meranie pokrytia kódu,
- *zdrojové súbory vnútri .jar súborov* – tiež nie sú pre pokrytie kódu významné.

Po inštrumentovaní tried, ktoré inštrumentovali pôvodne oba nástroje, som získala nové výsledky, ktoré sa už zhodujú. Tento experiment som uskutočnila na skupine testov `jboss-all-config-tests` a verzii AS 5.1.0 (Tabuľka 3.6). Výsledné pokrytie riadkov má hodnotu 31%.

3.3.4 Analýza testovacej sady JBoss AS

Ako som už napísala vyššie, testovacia sada je rozdelená do niekoľkých samostatných skupín. Testy využívajú testovací framework JUnit. Jednotlivé skupiny majú rôzne zameranie, ktoré vyplýva z ich názvu, a odlišujú sa od seba ako počtom testov a dĺžkou behu, tak aj pokrytím kódu, ktoré poskytujú. Podrobné informácie pre verziu JBoss AS 5.1.0 sú v Tabuľke 3.6. Čas v sekundách určuje len celkový čas potrebný na beh všetkých testov danej skupiny, nezahŕňa spúšťanie a vypínanie serverov alebo zápis výsledkov na disk.

Počet testov v jednotlivých skupinách sa pohybuje rádovo od jednotiek až po tisícky. Výnimku tvorí skupina `jboss-minimal-tests`, ktorá neobsahuje testy, iba spustí a vypne minimálnu konfiguráciu aplikačného servera.

Čas behu testov nezávisí priamo od ich počtu. Príkladom môže byť dvojica skupín testov `tests-jacc-security` a `tomcat-sso-clustered-tests`. Obe potrebujú približne rovnaký čas, ale počet testov druhej je až 20-krát vyšší.

Čo sa týka pokrytia kódu, vzťahy medzi hodnotami pre jednotlivé metriky sú rovnaké vo všetkých skupinách a zhodné aj s celkovými výsledkami pokrytia. Malé odchýlky môžu vzniknúť v tých skupinách testov, ktoré obsahujú viac testov, čo skončili s chybami. Také testy potom nemuseli prebehnúť celé a tým pádom nepokryli celý kód, ktorý by mali pokryť. Platí to napr. `pre tests-compatibility`.

Dve skupiny výrazne prevyšujú ostatné vo všetkých vyššie spomínaných kritériách, a to `jboss-all-config-tests` a `tests-clustering-all-stacks`. Tieto testy nie sú zamerané na konkrétnu časť aplikačného servera, ale testujú ho ako celok.

3.3.5 Optimalizácia testovacej sady

Beh všetých testov testovacej sady je náročný na čas a ten každou verziou vzrastá. Pri verzii JBoss AS 5.1.0 sú to už 3 hodiny a použitím nástroja Cobertura sa predĺži o viac ako hodinu. Preto by sme radi vedeli, či niektoré časti testovacej sady nie je možné vynechať s čo najnižšou stratou pokrytia kódu. Túto analýzu prevediem na troch verziách.

JBoss AS 3.2.8

Testovacia sada tejto verzie obsahuje len 8 skupín testov rozdelených do samostatných cieľov. Spustením testov po jednotlivých skupinách som zistila, že súčet takto získaných pokrytí je výrazne vyšší ako pokrytie namerané celou testovacou sadou, napr. v prípade pokrytia riedkov je to až o 64%. To znamená, že niektoré testy testujú rovnaké časti.

Najväčšiu časť pokrývajú testy cieľa `jboss-all-config-tests`, až 50% tried a 34% riadkov. Spustením zvyšných testov dosiahneme zvýšenie pokrytia len o 1% pre všetky

skupina testov	počet	čas	tried	metód	blokov	riadkov
jboss-minimal-tests	0	0	12%	7%	6%	6%
tests-jacc-security-allstarrole	2	3	22%	13%	9%	11%
tests-web-profile	2	7	20%	2%	9%	9%
tomcat-ssl-tests	6	7	16%	9%	7%	7%
tests-jts	246	7	20%	2%	9%	16%
tomcat-federation-tests	8	8	22%	13%	10%	11%
tests-bootstrap-dependencies	9	8	30%	18%	13%	15%
tests-springdeployer	4	9	22%	13%	10%	11%
tomcat-sso-tests	4	10	24%	14%	11%	12%
tests-binding-manager	3	19	26%	16%	12%	14%
tests-clustered-profileservice	14	42	27%	17%	13%	15%
tests-aop-scoped	68	49	27%	17%	13%	15%
tests-jbossmessaging-cluster	6	81	27%	16%	13%	14%
tests-jacc-security	192	90	30%	19%	16%	17%
tomcat-sso-clustered-tests	5	92	29%	18%	14%	16%
tests-compatibility	4	176	20%	1%	1%	1%
tests-clustered-classloader-leak	4	185	27%	17%	13%	14%
tests-jbossmessaging	281	259	25%	16%	12%	14%
tests-classloader-leak	16	337	27%	17%	13%	14%
jrmp-invoker-tests	71	364	26%	16%	13%	14%
pooled-invoker-tests	71	367	26%	16%	13%	14%
tests-profileservice	141	512	32%	21%	17%	18%
tests-clustering-all-stacks	753	2809	31%	20%	17%	18%
jboss-all-config-tests	1894	2891	48%	33%	29%	31%

Tabuľka 3.6: Prehľad jednotlivých skupín testov testovacej sady JBoss AS 5.1.0.

metriky. Z toho vyplýva, že pre získanie pokrytia takmer rovnajúceho sa pokrytiu, ktoré získame pri behu celej testovacej sady, postačí jeden cieľ. Tento cieľ však obsahuje výrazne vyšší počet testov, ktoré trvajú výrazne dlhšie, ako ostatné ciele v testovacej sade. Pokiaľ chceme skrátiť čas testovania, je vhodné tento cieľ vynechať.

Ciele, ktoré nepokrývajú takmer žiadny kód, ktorý by nebol pokrytý inými testami, je možné vynechať. Patria medzi ne `jboss-minimal-tests`, `tomcat-sso-clustered-tests`, `tomcat-ssl-tests` a `tomcat-sso-tests`. Ostatné ciele už takéto testy neobsahujú. Dostávame teda podmnožinu testovacej sady, ktorá obsahuje tri ciele z pôvodných ôsmich: `tests-security-manager`, `tests-clustering` a `test-example-binding-manager`. V nich obsiahnuté testy pokrývajú 36% tried, 24% metód, 18% blokov a 20% riadkov. Celkové množstvo týchto testov je 124 a čas potrebný na ich beh je 356 sekúnd.

JBoss AS 4.2.3

V tejto verzii je testovacia sada rozdelená na 19 skupín testov. Podobne ako pri verzii JBoss AS 3.2.8 platí, že niektoré časti kódu sú pokryté viacerými testami.

Najväčšiu časť pokrývajú znova testy cieľa `jboss-all-config-tests`, až 43% tried a takmer tretinu metód, blokov a riadkov. Tento cieľ je najväčší, obsahuje 2229 testov, ktoré

bežia 3352 sekúnd. Druhým najväčším cieľom je `tests-clustering-all-stacks`.

Podrobnou analýzou výsledkov som zistila, že cieľ `tests-clustering-all-stacks` je možné z testovania vynechať za cenu minimálnej straty pokrytia. Veľkú časť kódu pokrytého týmito testami pokrývajú testy v `tomcat-sso-clustered-tests`, ktorých je ale omnoho menej a vyžadujú kratšiu dobu pre svoj beh.

Cieľ `jboss-all-config-tests` pokrýva veľké množstvo takého kódu, ktorý nie je pokrytý žiadnymi inými testami, preto sa nedá vynechať. Rovnako aj ďalších 5 cieľov, ktoré ale pokrývajú aj kód pokrytý inými testami. Nakoniec dostaneme podmnožinu pôvodnej testovacej sady: `jboss-all-config-tests`, `tests-unified`, `test-example-binding-manager`, `tests-jacc-securitymgr`, `tomcat-sso-clustered-tests` a `tests-classloader-leak`. Obsahuje 3044 testov, ktoré bežia spolu 69 minút. Takto upravná testovacia sada pokrýva 45% tried, 32% metód, 28% blokov a 29% riadkov.

JBoss AS 5.1.0

Testovacia sada tejto verzie je najrozsiahlejšia z analyzovaných verzií, obsahuje až 26 skupín testov. Aj tu platí, že na dosiahnutie takmer rovnakého pokrytia, ako s celou testovacou sadou, netreba spúšťať celú testovaciu sadu.

Cieľ `jboss-all-config-tests` tentokrát pokrýva už 48% tried a viac ako tretinu metód, blokov a riadkov. Pre tento cieľ platí to isté, ako v predchádzajúcej verzii, nie je možné ho vynechať na rozdiel od druhého najväčšieho cieľa `tests-clustering-all-stacks`.

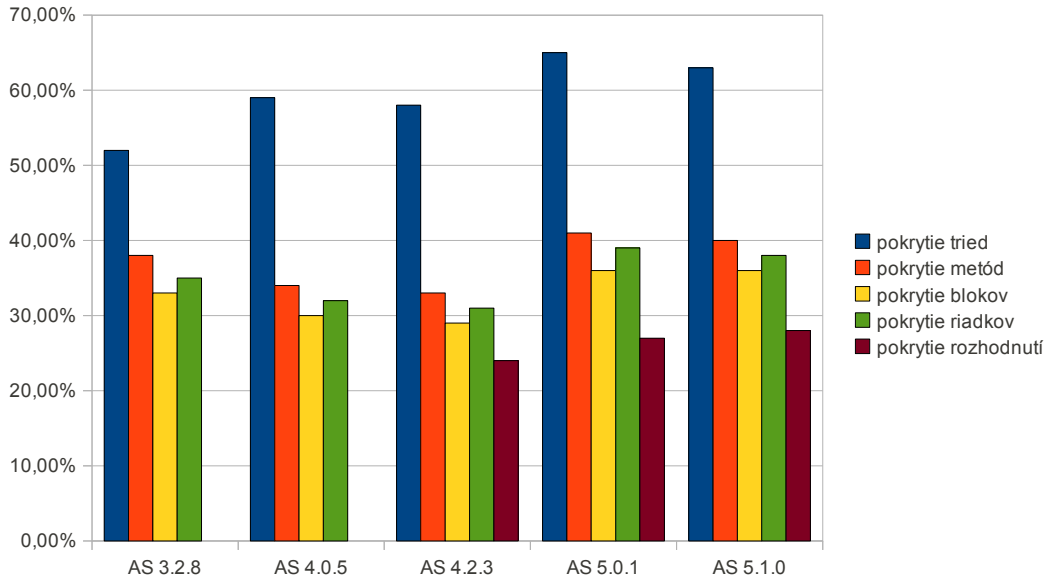
Minimálna podmnožina pôvodnej testovacej sady, ktorá pokrýva takmer rovnaké množstvo kódu, potom obsahuje nasledovné testy: `jboss-all-config-tests`, `tests-jts`, `tomcat-sso-clustered-tests`, `tests-compatibility`, `tests-profileservice` a `tests-aop-scoped`. Je v nej spolu 2358 testov, ktoré bežia spolu 69 minút a pokrývajú 60% tried, 38% metód, 33% blokov a 35% riadkov.

3.3.6 Porovnanie jednotlivých metrík pokrytia kódu

Emma a Cobertura poskytujú spolu 5 rôznych metrík pokrytia kódu. Každá metrika vyjadruje niečo iné a môže byť užitočná v inej situácii. Jednotlivé metriky je možné porovnať z hľadiska vypovedacej hodnoty a náročnosti zberu dát pre výpočet pokrytia.

Na Grafe 3.6 je vidieť, že metriky pokrytie metód, pokrytie blokov a pokrytie riadkov sú medzi sebou porovnateľné. Platí to nielen pre aplikačný server ako celok, ale aj pre jednotlivé balíčky či dokonca triedy. Pokrytie tried vo väčšine prípadov výrazne prevyšuje ostatné metriky. Je to spôsobené tým, že konkrétna trieda môže mať pokrytie tried buď 0% alebo 100%. V prvom prípade sú aj výsledky všetkých ostatných metrík 0%, pretože pokiaľ trieda nie je načítaná, nevykonáva sa ani žiaden jej kód. Na druhú stranu stopercentné pokrytie triedy je jednoducho dosiahnuteľné zavolaním jej konštruktora, ale vysoké pokrytie pre ostatné metriky vyžaduje hlbšie testovanie. Pokrytie rozhodnutí, na rozdiel od pokrytia tried, má vo všetkých prípadoch výrazne nižšie hodnoty ako ostatné metriky.

Skutočnosť, že na dosiahnutie rovnako vysokého pokrytia pre rôzne metriky je potrebné rôzne dôkladné testovanie, znamená, že nie všetky metriky sú rovnako vypovedajúce. Keď si napr. vyberieme ľubovoľný balíček, ktorý má pokrytie tried 100%, nemusí to znamenať, že je dôkladne testovaný. Ako príklad uvediem balíček *org.jgroups.demos.wb* vo verzii JBoss AS 5.1.0, ktorý má pokrytie tried 100%, ale pokryté sú len dva riadky v piatich blokoch a v jednej metóde. Tento balíček teda nie je vôbec testovaný a podobne sú na tom mnohé ďalšie balíčky so stopercentným pokrytím tried. Pokrytie tried je viac užitočné v prípade, kedy potrebujeme identifikovať triedy, ktoré nie sú počas testov vôbec nahrané do JVM.



Obrázok 3.6: Graf pokrytia podľa jednotlivých metrík a verzie aplikačného servera.

Pokrytie metód je spoľahlivejšie kritérium ako pokrytie tried. Pokiaľ sú všetky metódy danej triedy pokryté, je vysoká pravdepodobnosť, že je trieda testovaná. Svedčí o tom väčšina tried s pokrytím metód 100%. Ak aj nie sú pokryté všetky riadky či bloky danej metódy, jedná sa o menej časté vetvy podmieňovacích príkazov, ošetrovanie výnimiek apod. Príkladom môže byť trieda *EntityBeanCacheBatchInvalidatorInterceptor.java* v balíčku *org.jboss.cache.invalidation.triggers*, ktorá má pokrytie metód 100%, pokrytie blokov 77% a pokrytie riadkov 83%. Pohľadom do zdrojového kódu v správe o pokrytí zistíme, že neboli vykonané len dve alternatívne vetvy podmieňovacieho príkazu a zvyšok kódu je úplne pokrytý. Rovnako aj hodnoty tejto metriky pre ostatné triedy nám dávajú pomerne dobrý obraz o tom, nakoľko je skúmaná trieda testovaná.

Podobne sú na tom aj pokrytie základných blokov a pokrytie riadkov. Tieto hodnoty sú pre väčšinu tried veľmi blízke. Obe sú vhodné na zisťovanie miery celkového pokrytia kódu. Rozdiel medzi nimi je, že kým 100% pokrytie blokov implikuje 100% pokrytie riadkov, naopak to nemusí platiť. Spoločnou nevýhodou týchto metrík je, že zbieranie dát pre ne je náročnejšie ako pre pokrytie tried a metód.

Pokrytie rozhodnutí je vo všetkých prípadoch najnižšie. Dôvod je ten istý, ako je v prípade, že trieda má stopercentné pokrytie metód, ale ostatné metriky ukazujú nižšie hodnoty. Teda spracovanie výnimiek a veľký počet vetiev programu, ktoré nie je potrebné vykonávať. Často ani nie je možné dosiahnuť pokrytie rozhodnutí 100%.

Každá z analyzovaných metrík má svoje výhody a nevýhody. Najlepší obraz o pokrytí podávajú zrejme pokrytie riadkov a základných blokov za cenu najväčších nákladov na zbieranie dát počas behu testov. Naopak pokrytie tried nezodpovedá celkovej miere, do akej je aplikácia testovaná. Všeobecne je však najľahšie počítať práve toto pokrytie tried, pretože v aplikácii býva niekoľkonásobne menej tried ako ostatných jednotiek.

Z hľadiska aplikačného servera JBoss je nutné nájsť vhodný kompromis medzi týmito dvomi kritériami, ktorým môže byť pokrytie metód. Má relatívne jednoduché zbieranie dát a dáva veľmi slušný prehľad o testovaní.

3.4 Zhodnotnie použitých nástrojov

Nástroje, ktoré som v tejto práci použila, spĺňajú všetky kritériá, ktoré boli spomenuté na začiatku tejto kapitoly. Aj napriek tomu existujú dodatočné kritériá, na základe ktorých je možné ich porovnať po ich použití na aplikačný sever. Sú to vlastnosti ako nároky, užívateľská prívetivosť či kvalita generovaných správ.

Čo sa týka pamäťových nárokov, Emma aj Cobertura sú na tom podobne. Oproti neinštrumentovanému aplikačnému serveru nepotrebuje inštrumentovaná verzia znateľne väčšie množstvo pamäte.

Časová náročnosť nástrojov sa však už dosť líši. Na čas behu samotných testov nemá zber dát pre analýzu pokrytia kódu nijaký význam, avšak počas testovania je niekoľkokrát spúšťaný aplikačný server. Navyše Cobertura zapisuje výsledky do súboru *cobertura.ser* po skončení každého testu, čo spôsobuje ďalšie spomalenie. Emma na rozdiel od Cobertury nijak viditeľne nespomalí beh aplikačného servera. Testovanie so zberom dát pre Emmu sa predĺži len o 1%, pre Coberturu až o 43% celkového času.

Užívateľská prívetivosť, oboch nástrojov je prijateľná. Vyžadujú približne rovnako náročnú konfiguráciu. Nevýhodou nástroja Emma je, že výsledky merania pokrytia riadkov neposkytuje v prípade, že niektoré triedy neobsahujú ladiace informácie. Tieto triedy potom musíme vylúčiť z merania pokrytia, ak chceme získať aj pokrytie riadkov. Naopak jeho veľkou výhodou oproti Cobertury je, že ukladá statické a dynamické dáta do dvoch rôznych súborov. To môže byť užitočné pri opakovanom inštrumentovaní aplikačného servera s rôznym výberom tried pre inštrumentáciu. V takom prípade je nutné sputiť testy len raz s inštrumentovaným celým aplikačným serverom. Po vynechaní niektorých tried z inštrumentácie už nie je potrebné spúšťať znova testy, stačí spustiť inštrumentáciu a vygenerovať správu o pokrytí použitím pôvodných *.ec* súborov.

Schopnosť generovať prehľadnú správu o pokrytí je v prípade takého rozsiahleho projektu, akým je aplikačný server, veľmi užitočná. Najprehľadnejším formátom je jednoznačne HTML, pretože poskytuje nielen textový, ale aj grafický náhľad na podávané informácie. Oba použité nástroje takúto správu generujú. V čom sa ale odlišujú, sú určité detaily, ktoré robia správu od Cobertury lepšou. Medzi ne patria napr. možnosť usporiadania tried a balíčkov podľa ľubovoľného stĺpca tabuľky či možnosť zobrazenia všetkých tried projektu v jednom zozname. Emma zoraďuje informácie podľa hodnôt pokrytia základných blokov a pokiaľ chceme vidieť informácie o konkrétnej triede, musíme sa k nej dostať cez príslušný balíček. Ak má správa generovaná Coberturou k dispozícii zdrojový súbor triedy, zobrazuje nielen to, či daný riadok je alebo nie je pokrytý, ale aj počet, koľkokrát bol počas testov vykonaný. Na druhú stranu Emma dokáže rozoznať aj čiastočne pokryté riadky a v zdrojovom súbore ich tiež rozlišuje.

Kapitola 4

Záver

Cieľom práce bolo navrhnúť a realizovať postup pre získanie pokrytia kódu aplikačného servera JBoss a zistené výsledky analyzovať. Na základe určených kritérií som vybrala nástroje Emma a Cobertura, ktoré poskytujú aj rôzne, aj spoločné metriky pokrytia kódu. Výsledkom je detailný postup, ako tieto nástroje použiť a správy o pokrytí, ktoré generujú. Tieto správy majú HTML aj XML formát. HTML formát poskytuje prehľadné informácie, kým XML formát je vhodnejší pre prípadne ďalšie použitie a analýzu.

Vývoj pokrytia kódu bolo možné ukázať len pomocou nástroja Emma, pretože narozdiel od Cobertury dokáže pracovať so všetkými verziami JBoss AS. Ako sa aplikačný server vyvíjal, vyvíjala sa aj jeho testovacia sada. Aj napriek miernemu poklesu najmä vo verzii 4.0.5 pokrytie kódu pomaly stúpalo. Najvyššie hodnoty pre všetky dostupné metriky dosiahlo vo verzii 5.0.1.

Nástroje na analýzu pokrytia kódu, ktoré som v práci použila, sú značne odlišné. Oba je síce možné integrovať do zostavovacieho procesu pomocou aplikácie Ant, poskytujú statickú inštrumentáciu tried a generujú správu o pokrytí vo formáte HTML aj XML, výrazne sa ale odlišujú v meraných metrikách pokrytia kódu, časových nárokoch, kvalite generovanej správy či detailoch pri použití.

Pre každý projekt môže byť výhodnejšie použiť iný nástroj na analýzu pokrytia kódu. V prípade aplikačného servera a projektov podobne veľkých rozmerov s rozsiahlou testovacou sadou sa javí vhodnejšie použitie nástroja Emma. Dôvodov je niekoľko: väčšie množstvo metrík pokrytia kódu, časová aj pamäťová náročnosť je minimálna, ukladanie statických a dynamických dát oddelene je veľmi výhodné a urýchľuje prácu.

Emma a Cobertura poskytujú jedinú spoločnú metriku pokrytia kódu, ktoru je pokrytie riadkov. Výsledky získané týmito nástrojmi sa však odlišujú. Dôvodom je, že Emma neinštrumentuje a nezbiera dáta pre niektoré súbory. Príkladom sú anonymné vnútorné triedy, ktoré Emma nedokáže rozoznať. Po vylúčení z inštrumentácie týchto tried aj pri nástroji Cobertura sa už výsledky pokrytia kódu zhodovali.

Analýzou získaných informácií z hľadiska rôznych metrík pokrytia kódu som dospela k záveru, že pre taký rozsiahly projekt, akým je aplikačný server, nie je jednoduché vybrať ideálnu metriku. Pokrytie tried môže byť vhodným kompromisom medzi nízkou réžiou na získanie dát a poskytovaní čo najvernejšieho obrazu o pokrytí kódu.

Testovacia sada JBoss AS sa skladá z niekoľkých skupín testov. Spustením jednotlivých skupín samostatne som získala aj čiastočné hodnoty pokrytia a zistila som, že medzi počtom testov, ich časovou náročnosťou a množstvom kódu, ktorý pokrývajú, nie je priamy vzťah. Rovnako som zistila, že niektoré testy pokrývajú rovnaký kód ako testy v inej skupine. Niektoré skupiny testov je možné z testovania vynechať za cenu minimálnej alebo žiadnej

straty pokrytia kódu. Takto sa dá ušetriť čas behu testov, ktorý predstavuje v závislosti od verzie a použitého nástroja niekoľko hodín.

Všetky ciele práce boli splnené, výsledky budú použité firmou Red Hat pre analýzu nedostatkov v testovacej sade a pre jej rozšírenie. Zároveň budú vybrané metriky na základe dodaných podkladov pravidelne sledované a vyhodnocované. V súčasnej dobe vzniká úplne nová verzia aplikačného servera JBoss AS 7, ku ktorej je vytváraná nová testovacia sada. Veľmi významné môže byť zistenie, že väčšina kódu bola otestovaná len asi polovicou testov testovacej sady. Táto skutočnosť by mala byť pri novej testovacej sade kontrolovaná, aby sa nevytvárali duplicitné testy.

Literatúra

[1] Astels, D.: *Test-Driven Development : A Practical Guide*. Prentice-Hall, 2003, iSBN 0-13-101649-0.

[2] Bron, A.; Farchi, E.; Magid, Y.; aj.: Applications of synchronization coverage. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, New York, NY, USA: ACM, 2005, s. 206–212, iSBN 1-59593-080-9.
URL http://doi.acm.org/10.1145/1065944.1065972

[3] Cornett, S.: Code Coverage Analysis [online]. 2010 [cit. 4.2.2011].
URL http://www.bullseye.com/coverage.html

[4] Eckel, B.: *Thinking in Java*. New Jersey, Prentice-Hall, 1998, iSBN 01-365-9723-8.

[5] Eric Jendrock, e. a.: The Java EE 5 Tutorial [online]. 2010 [cit. 31.12.2010].
URL http://java.sun.com/javaee/5/docs/tutorial/doc/index.html

[6] Eric Jendrock, e. a.: Java EE 6 Tutorial : Basic Concepts [online]. November 2010 [cit. 31.12.2010].
URL http://download.oracle.com/javaee/6/tutorial/doc/index.html

[7] Hall, M.: *Java : servlety a stránky JSP*. Praha, Neocortex, 2001, iSBN 80-863-3006-0.

[8] Hynar, M.: *JAVA – nástroje*. Praha, Neocortex, 2004, iSBN 80-86330-16-8.

[9] J. Thomas, e. a.: *Java Testing Patterns*. Indianapolis, Wiley Publishing, Inc., 2004, iSBN 0-471-44846-X.

[10] Jamae, J.; Johnson, P.: *JBoss in Action*. Greenwich, Manning Publications Co., 2009, iSBN 978-1-933988-02-3.

[11] Jameson, R.: Clover Tutorials [online]. 27.8.2007 [cit. 3.5.2011].
URL http://confluence.atlassian.com/display/CLOVER/Clover+Tutorials

[12] Kolektív autorov: CodePro AnalytiX Evaluation Guide [online]. 2010 [cit. 3.5.2011].
URL
http://google-web-toolkit.googlecode.com/files/CodePro-EvalGuide.pdf

[13] Lindholm, T.; Yellin, F.: *The Java virtual machine specification*. Boston, Addison-Wesley, 2003, iSBN 02-014-3294-3.

[14] Mukhar, K.; Zelenak, C.: *Beginning Java EE 5 : From novice to professional*. Apress, 2006, iSBN 15-905-9470-3.

[15] Nir-Buchbinder, Y.; Ur, S.: Multithreaded unit testing with ConTest [online]. 3. 3. 2005 [cit. 27.3.2011].
URL http://www.ibm.com/developerworks/java/library/j-contest.html

[16] Roubtsov, V.: EMMA Reference Manual [online]. 2006 [cit. 3.5.2011].
URL http://emma.sourceforge.net/reference/reference.html

[17] Smart, J. F.: *Java Power Tools*. Sebastopol, O'Reilly, 2008, iSBN 978-0-596-52793-8.

[18] Tom Marrs, S. D.: *JBoss at Work : a practical guide*. Sebastopol, O'Reilly, 2006, iSBN 978-059-6007-348.

[19] Young, M.; Thiemel, A.: *XML : krok za krokem*. Brno, Computer Press, 2006, iSBN 80-251-1070-2.

Dodatok A

Obsah CD

Priložené CD obsahuje:

- súbor *README* popisujúci adresárovú štruktúru CD,
- PDF verziu tejto práce,
- skripty *code-coverage.xml* pre každú analyzovanú verziu JBoss AS,
- správy o pokrytí generované použitými nástrojmi.